A

Verification of database-driven systems via amalgamation

Mikołaj Bojańczyk [1], Univ. of Warsaw
Luc Segoufin, INRIA and ENS Cachan
Szymon Toruńczyk [1], Univ. of Warsaw

We describe a framework for static verification of systems that base their decisions upon queries to databases. The database is specified using constraints, typically a schema, and is not modified during a run of the system. The system is equipped with a finite number of registers for storing intermediate information from the database and the specification consists of a transition table described using quantifier-free formulas that can query the database and the registers.

Our main result concerns systems querying XML databases – modeled as data trees – using quantifier-free formulas with predicates such as the descendant axis or comparison of data values. In this scenario we show an ExpSpace algorithm for deciding reachability.

Our technique is based on the notion of amalgamation and is quite general. For instance it also applies to relational databases (with an optimal PSpace algorithm).

We also show that minor extensions of the model lead to undecidability.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Logic, automata

Additional Key Words and Phrases: Database-driven Systems, Register Automata, Amalgamation, Fraïssé classes

Contents

1 Introduction **2**

2 Preliminaries **4**

3 Database-driven systems **4**
3.1 Database-driven register automata 4
3.2 Database-driven Büchi register automata 6
3.3 Database-driven register tree automata 6
3.4 Database-driven pushdown automata 7
3.5 The emptiness problem 7
3.6 Overview of the algorithm 8

4 Homogeneity, amalgamation and Fraïssé classes **8**
4.1 Limits of classes of databases 8
4.2 Homogeneity 9
4.3 Amalgamation 10
4.4 Fraïssé classes and their limits 11

5 Evaluating fixpoint logics over homogeneous structures **12**
5.1 Syntax and semantics of transitive closure and least fixpoint logics 12
5.2 Evaluation in homogeneous structures 13

6 Complexity of emptiness for database-driven systems **15**
6.1 Definable sets and structures 15

7 Semi-Fraïssé classes **18**
7.1 Classes closed under amalgamation and embeddings 18

7.2 Semi-Fraïssé classes 19
7.3 Data values 20
7.4 HOMs are Semi-Fraïssé 22

8 XML documents and regular tree languages 23

9 Regular Tree Languages 25
9.1 Tree automata and their components 25
Horizontal components26
9.2 The class $\mathcal{C}$ 26

10 Undecidable models 32
10.1 Child and sibling axes 32
The sibling axis33
10.2 Rules that are not existential 33
10.3 Data tree patterns 34

11 Conclusions 36

1. INTRODUCTION

In this paper we describe a framework for static verification of *database-driven* systems. Such a system bases its decisions upon queries to databases. Typical examples are web services, web applications, or data-centric business processes. These systems can be complex and error prone. Computer-aided static analysis can improve their robustness and correctness.

In order to perform static analysis, the behavior of the database-driven system is specified in a suitable formalism; the desired properties of its executions are also specified in a suitable formalism. The computer then automatically checks whether all runs of the system verify the expected properties.

As advocated in [Vianu 2009], classical software verification techniques have serious limitations when applied to such systems – the main reason is that they abstract away data values, resulting in serious loss of semantics for both the system and the properties being verified.

For this reason, several specific formalisms have been designed allowing meaningful specification of relational database-driven systems. See for instance [McIlraith et al. 2001; Deutsch et al. 2006; Deutsch et al. 2009; Deutsch et al. 2007; Segoufin and Toruńczyk 2011]. As demonstrated in [Segoufin and Toruńczyk 2011], for these scenarios, the system can be described using a register automaton whose transition rules are quantifier-free first-order formulas querying the database and the registers. The correctness criterion for executions of the system is specified using a language mixing queries to the database and temporal behavior that can easily be translated into the same register automata model. Altogether the static analysis problem boils down to testing the existence of a database such that the register automaton has an accepting run driven by that database.

In this paper we develop general techniques for testing reachability of such automata models. These techniques encompass the examples cited above concerning relational databases, but also apply to XML databases, and – in general – to any kind of structures having "good" model properties.

Following [Segoufin and Toruńczyk 2011], we specify database-driven systems using transition rules controlling their workflow. Each such rule may be based on the result of quantifier-free queries to the database. The database is not fixed and may vary from run to run. It is however restricted to range over a certain class of databases typically specified using a schema and possibly several other constraints. Moreover the system has only read access to the database and the database does not change during a run.

To give an idea of the setting we are dealing with, let us describe a toy example of a database-driven system $\mathcal{S}$ that fits into our framework. The system $\mathcal{S}$ is equipped with one register capable of storing nodes of XML documents. We specify the transitions of $\mathcal{S}$ as follows:

> *The node stored in the register after the transition is a descendant of the node stored in the register before the transition and the attribute* `a` *of both nodes (before and after) contains the same data value.*

Furthermore, we specify that in the initial configuration of the system, the register stores the root of the tree, and in an accepting configuration, it must store some leaf of the tree. Note that the transitions of the system do not modify the database.

We are interested in the following question: is there some XML document t such that the described system has an accepting run driven by t? We may ask more detailed questions: is there some XML document t satisfying a certain XML schema such that the described system has an accepting run driven by t?

In general our goal is to give an algorithm for the following problem, parametrized by a class $\mathcal{C}$ of databases.

- **Input.** A database-driven system.
- **Output.** Does the system have some finite accepting run driven by some database in $\mathcal{C}$?

We show that if the class $\mathcal{C}$ of databases satisfies a certain model-theoretic assumption – namely, it is a computable *Fraïssé class* – then there exists an algorithm for the described problem.

Our main technical result shows that many natural classes of databases are Fraïssé. Examples include: all databases over a given relational schema, three-colorable graphs (more generally, any property of databases expressed as a Constraint Satisfaction Problem), XML documents viewed as data trees satisfying a given XML schema (more generally, any property of trees recognized by a tree automaton).

The most interesting and most difficult result is the XML case. In this scenario the database is an XML document that must verify a certain XML schema. The system can query the XML document using the descendant axis, the document order and the closest common ancestor relation. It can also test equality or inequality between attribute values. Our technique shows that in this setting the above problem is decidable in ExpSpace.

In the setting of relational databases, we derive from our technique an optimal PSpace decision procedure.

We also show how extending the expressive power of these systems quickly leads to undecidability. For instance, in the XML setting, allowing the system to use the sibling axis or the child axis in queries leads to undecidability.

Comparison with previous work. The model described in this paper generalizes the previous existing models of automata introduced for relational database-driven systems [McIlraith et al. 2001; Deutsch et al. 2006; Deutsch et al. 2009; Deutsch et al. 2007; Segoufin and Toruńczyk 2011]. In particular, the domain can be linearly ordered and the specification of the database-driven system may use this order within the quantifier-free formulas. In this paper we notice that the key is the Fraïssé property, which holds for linear orders, and show that the XML setting is also Fraïssé.

In terms of results, this paper generalizes all the previous known results concerning the existence of finite runs. As shown in several of the above cited papers, the existence of infinite runs lead to additional challenges which are not solved by the Fraïssé property alone. However, in most of the practical cases mentioned here the existence of infinite runs can be reduced to the existence of finite runs using a Ramsey argument as described in [Segoufin

and Toruńczyk 2011]. Our results only encompass the simple special case of infinite runs where all data values come from a finite domain.

This paper is the journal version of [Bojańczyk et al. 2013]. Since the conference version we have improved the complexity analysis by going from data complexity to combined complexity. The proof have also been inlcuded.

2. PRELIMINARIES

We model databases as finite structures over finite schemas containing relation and function symbols. We use notions which are standard in model theory (see [Hodges 1997] for a reference), sometimes using terminology from database theory to denote them. We briefly recall the relevant notions below.

A *schema* Σ is a finite set of relation symbols and function symbols, each with a given arity (0-ary function symbols are *constant* symbols). A *model*, or *structure*, $\mathbb{A}$ over a schema Σ is a set $\mathrm{dom}(\mathbb{A})$ – the *domain* of $\mathbb{A}$ – together with an interpretation $s^{\mathbb{A}}$ for each symbol $s \in \Sigma$ as a relation or function over the domain of an appropriate arity, as described by the schema. A structure is said to be *finite* if its domain is finite. A *database* is a finite structure over a given schema.

By *substructure* we always mean in this paper an *induced substructure*, i.e. a restriction of the initial structure to a subset of its domain, which is closed under the function symbols from the schema.

A structure is said to be *generated* by a subset S of its domain if the structure has no proper substructure whose domain also contains S. A structure is called *n-generated* if its domain has a subset of size n that generates it, and is called *finitely generated* if it is n-generated for some number n.

A *homomorphism* from a structure $\mathbb{A}$ to a structure $\mathbb{B}$, is a mapping $h : \mathrm{dom}(\mathbb{A}) \to \mathrm{dom}(\mathbb{B})$ that preserves the relations and functions from Σ, i.e. $(a_1, \ldots, a_k) \in R^{\mathbb{A}}$ implies $h(a_1, \ldots, a_k) \in R^{\mathbb{B}}$, and $h(f^{\mathbb{A}}(a_1, \ldots, a_k)) = f^{\mathbb{B}}(h(a_1, \ldots, a_k))$, for all tuples $a_1, \ldots, a_k$ of elements of $\mathrm{dom}(\mathbb{A})$ and function/relation symbols of arity k. An *isomorphism* is a bijective homomorphism whose inverse mapping is also a homomorphism. An *automorphism* is an isomorphism from $\mathbb{A}$ to itself. Finally an *embedding* is a mapping h that is an isomorphism onto the substructure induced by the image of h.

We assume familiarity with first-order logic. We write $\mathbb{A} \models_{\mathrm{val}} \varphi$ to express the fact that a first-order formula φ holds in the structure $\mathbb{A}$ with the valuation val for its free variables.

3. DATABASE-DRIVEN SYSTEMS

A database-driven system is a transition system whose decisions are based upon querying a database. In this paper we consider models of automata equipped with registers for storing data values and whose transitions are guarded by quantifier-free formulas that query the database and update the content of the registers. More specifically we consider the following systems: finite state register automata, Büchi automata, tree automata and pushdown automata. However our results could work for other systems.

3.1. Database-driven register automata

A *database-driven register automaton* over a schema Σ is given by the following components.

- A finite set of *control states* $Q = \{p, q, \ldots\}$
- A finite set of *registers* $X = \{x, y, \ldots\}$
- A subset of *initial* states $I \subseteq Q$
- A subset of *accepting* states $F \subseteq Q$
- Finitely many *transition rules* of the form:

$$p \xrightarrow{\delta} q \tag{1}$$

where p, q are control states and δ is a quantifier-free first-order formula over the schema Σ whose variables are from two disjoint copies of the registers, called X_{old} and X_{new}. The formula δ is called the *guard* of the transition and relates the values of the registers before and after the transition.

A *configuration* of the system consists of: a structure over the schema Σ, a control state, and a valuation of the registers which maps each register to an element of the domain of the structure.

A transition rule as in (1) is said to *connect* one configuration with another if: both configurations have the same structure, the state in the first configuration is p, the state in the second configuration is q, and the guard δ holds in the structure with a valuation whose restriction to the variables X_{old} corresponds to the valuation of the registers of the first configuration, and whose restriction to the variables X_{new} corresponds to the valuation of the registers of the second configuration.

A *run* of the system is a sequence of configurations that begins in a configuration with an initial state, and where every two consecutive configurations are connected by some transition rule. In the case of database-driven register automata we are interested in finite runs. A finite run is *accepting* if the control state in its last configuration is accepting. Since a transition rule cannot change the structure between configurations, every run has a unique associated structure that is shared by all configurations in the run. We say that the run is *driven by* that structure. Note that different runs of the same system may be driven by different structures.

The problem we are interested in concerns runs driven by a database, i.e. a finite structure. However our method uses an intermediate infinite structure as explained in Section 4. This is why our definition of configuration and run allows for arbitrary structures.

Example 3.1. Consider directed graphs, where some of the nodes are colored red, and the remaining nodes are white. This corresponds to a schema with one binary edge predicate E and one unary predicate *red*.

We describe a database-driven register automata whose accepting runs trace odd-length cycles of red nodes. The system has the following components.

- The control states are $\{\mathit{start}, q_0, q_1, \mathit{end}\}$. The initial state is *start* and the accepting state is *end*.
- The registers are x, y.
- There is a transition rule $q_0 \xrightarrow{\delta} q_1$, where the guard δ is

$$(x_{\text{old}} = x_{\text{new}}) \wedge E(y_{\text{old}}, y_{\text{new}}) \wedge \mathit{red}(y_{\text{new}}).$$

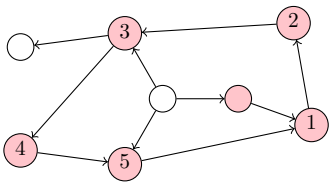
  There is also a transition rule $q_1 \xrightarrow{\delta} q_0$, with the same guard. This means that the system alternates between the states q_0 and q_1, each time moving the content of register y along an edge to some red node (the content of register x stays in place).

  There is a transition rule $\mathit{start} \xrightarrow{\alpha} q_0$, where the guard α is

$$x_{\text{old}} = x_{\text{new}} = y_{\text{old}} = y_{\text{new}}$$

  There is also a transition $q_1 \xrightarrow{\alpha} \mathit{end}$, with the same guard. This means that, in order to exit the initial state, both registers need to point to the same vertex; likewise, in order to enter the accepting state, both registers need to point to the same vertex.

Here is an example run of the system. The run is driven by the database that is the graph $\mathbb{G}$ depicted below. The nodes of the graph are colored red or white; the numbers are not part of the database, they are used to identify the nodes.



An accepting run of the system, driven by $\mathbb{G}$ is:

$$(\mathbb{G}, \mathit{start}, [1,1]) \to (\mathbb{G}, q_0, [1,1]) \to (\mathbb{G}, q_1, [1,2]) \to (\mathbb{G}, q_0, [1,3]) \to \\ \to (\mathbb{G}, q_1, [1,4]) \to (\mathbb{G}, q_0, [1,5]) \to (\mathbb{G}, q_1, [1,1]) \to (\mathbb{G}, \mathit{end}, [1,1]),$$

where $[i,j]$ denotes the valuation that maps x to the node marked with i and y to the node marked with j.

In general, the described system has an accepting run driven by a graph $\mathbb{G}$ if and only if there is a cycle in $\mathbb{G}$ of odd length, consisting only of red nodes.

3.2. Database-driven Büchi register automata

We also consider infinite runs with a Büchi acceptance condition. The syntax and configuration of a *database-driven Büchi register automaton* is the same as for a database-driven register automaton, only the notion of accepting run is changed. We require that the infinite run involves only finitely many different values for the registers. This condition is trivially met when the run is driven by a finite database but may not be if the underlying structure is infinite. Finally an infinite run is called accepting if it visits an accepting control state infinitely often, possibly with different register valuations.

An alternative definition would say that an infinite run is accepting if there is some configuration with an accepting control state that appears infinitely often, with the same register valuations. As long as we are interested in finite databases, the alternative definition is equivalent, since there are finitely many possible configurations over a given database. More precisely, for every finite database, a run driven by the database is accepting under one definition if and only if it is accepting run under the other definition.

Note that our technique can only deal with the simple case of infinite runs containing only finitely many different values for its registers. Unlike [Vianu 2009; Deutsch et al. 2007] and [Segoufin and Toruńczyk 2011] we don't consider values ranging from an infinite numerical domain such as $\langle \mathbb{N}, < \rangle$ or $\langle \mathbb{Q}, < \rangle$.

However, for the specific case of runs driven by $\langle \mathbb{N}, < \rangle$ or $\langle \mathbb{Q}, < \rangle$ (actually any linear order), the existence of infinite runs can be reduced to the existence of finite runs using a Ramsey argument as described in [Segoufin and Toruńczyk 2011].

3.3. Database-driven register tree automata

We also consider (finite) tree automata. The syntax of a *database-driven register tree automaton* is the same as for a database-driven register automaton, except for the transition relation that contains now rules of the form:

$$p_1, p_2 \xrightarrow{\delta} q \tag{2}$$

where p_1, p_2, q are control states and δ is a quantifier-free first-order formula over the schema Σ whose variables are from three disjoint copies of the registers, called $X_{\text{old},1}$, $X_{\text{old},2}$ and X_{new}. The formula δ is called the *guard* of the transition and relates the values of the registers before and after the transition.

A configuration is defined as before and three configurations are connected by a transition as in (2) if: all configurations have the same structure, the state in the first and second configurations are p_1, p_2, the state in the third configurations is q, and the guard δ holds

in the structure with a valuation whose restriction to the variables $X_{\text{old},1}$ and $X_{\text{old},2}$ corresponds to the valuation of the registers of the first and second configurations, and whose restriction to the variables X_{new} corresponds to the valuation of the registers of the third configuration.

A run of such an automaton is then a finite binary tree labeled by configurations, such that the configurations at the leaves have initial states, the configuration at the root has an accepting state, and at any internal (non-leaf) node, the configuration is connected with the configurations of its children by some rule in the transition relation.

3.4. Database-driven pushdown automata

We also consider pushdown automata. The idea is to extend a register automaton with a pushdown stack, where tokens of the stack have registers that store elements of the structure. A *database-driven pushdown automaton* over a schema Σ is given by the following components.

- A finite set of *control states* Q
- A finite set of *control registers* X
- A finite set of *stack labels* Γ
- A finite set of *stack registers* Y
- A subset of *initial* states $I \subseteq Q$
- A subset of *accepting* states $F \subseteq Q$
- Finitely many *transition rules* which can have two possible forms:

$$\underbrace{(p,a) \xrightarrow{\delta} q}_{\text{pop}} \qquad \text{or} \qquad \underbrace{q \xrightarrow{\delta} (p,a)}_{\text{push}} \tag{3}$$

  where p, q are control states, a is a stack label and δ is a quantifier-free first-order formula over the schema Σ with free variables included in $Y \cup X_{\text{old}} \cup X_{\text{new}}$, with X_{old} and X_{new} being two copies of X. The formula δ is called the *guard* of the transition.

A *stack token* over a structure is a stack label and a valuation of the stack registers to elements of the domain of the structure. A *stack* over a structure is defined to be a sequence of stack tokens. A configuration of a database-driven pushdown automaton consists of a structure, a control state, a valuation of the control registers, and a stack. Configurations are connected by transition rules as in database-driven register automata, except that the pop transition can also access the stack label and register valuation of the popped token via the variables Y of the guard, while the push transition can access the stack label and valuation of the pushed token via the variables Y of the guard. An accepting run is a sequence of configurations that begins in an initial control states and ends in an accepting control state.

3.5. The emptiness problem

For each kind of database-driven systems we are interested in the emptiness problem, which is parametrized by a class $\mathcal{C}$ of databases (i.e. finite structures) over a common schema Σ, and called *emptiness of database-driven systems over* $\mathcal{C}$.

- **Input.** A database-driven system over Σ.
- **Output.** Does the system have some accepting run driven by some database in $\mathcal{C}$?

Actually, in some of our results also a finite description of the class $\mathcal{C}$ will be given on input.

The following observation shows that the problem is PSPACE-hard for almost any choice of parameter $\mathcal{C}$.

LEMMA 3.2. *The emptiness problem for database-driven systems over $\mathcal{C}$ is* PSPACE*-hard if $\mathcal{C}$ contains at least one database with at least two elements.*

PROOF. Assuming that there are at least two possible elements in the database, a bit string of length n can be encoded in a valuation of registers numbered from 0 to n. The value of bit $i \in \{1, \ldots, n\}$ is 1 if register i is equal to register 0, and false otherwise.

We reduce from the termination problem for Turing Machines working in linear space. We use a register y to store one element of the domain. This value is decided via the initial transition and remains unchanged during the run of the system. We use n registers $x_i, \cdots, x_n$ for encoding the content of the tape of the Turing Machine: cell i of the tape is equal to 1 iff the value of x_i is equal to the value of y. Using quantifier free formulas of length n, it is easy to simulate the transitions of a Turing Machine working in linear space. □

Existential guards. Before describing our results in more details, we point out that replacing quantifier-free formulas by existential formulas in the guards when specifying the system does not affect the expressive power nor the decidability results, as quantified variables can be simulated by using extra registers and nondeterminism. However, as we shall show later on, further extensions of the guards, such as boolean combinations of existential formulas, break decidability.

3.6. Overview of the algorithm

We now give a high-level overview of our algorithm for the emptiness problem for database-driven systems, on the example of database-driven register automata. If $\mathcal{A}$ is a database-driven register automaton and $\mathbb{C}$ is a possibly infinite input structure over the schema of $\mathcal{A}$, then define the *configuration structure of $\mathcal{A}$ over $\mathbb{C}$*, denoted by $\mathrm{Conf}(\mathbb{C}, \mathcal{A})$, to be the following structure. The domain is the configurations (i.e. recall that a configuration is a control state plus a register valuation), there is a binary predicate for the one step transition relation, and there are unary predicates for initial and accepting configurations. The two main steps in the algorithm are:

(1) If a class of databases $\mathcal{C}$ is sufficiently well-behaved, namely it is a Fraïssé class, then there exists a limit structure $\mathbb{C}$, usually infinite, with the following property. For every database-driven register automaton $\mathcal{A}$, there exists an accepting run of $\mathcal{A}$ driven by some database in $\mathcal{C}$ (a database is finite by definition) if and only if there exists an accepting run driven by the limit structure.
(2) Whether or not $\mathcal{A}$ has an accepting run driven by the limit of the structure is a reachability problem which can be expressed by a formula of transitive closure logic evaluated over the configuration structure $\mathrm{Conf}(\mathbb{C}, \mathcal{A})$. Assuming further properties on $\mathcal{C}$, this configuration structure has decidable model checking for transitive closure logic.

A similar approach is used for Büchi automata, or tree automata, or pushdown automata. The only difference is that for tree automata and pushdown automata, instead of transitive closure logic we need to use the more powerful least fixpoint logic, at the cost of worse complexity.

4. HOMOGENEITY, AMALGAMATION AND FRAÏSSÉ CLASSES

In this section, we define Fraïssé classes of finite structures (or databases) and their limits. For Fraïssé classes, the proof strategy described in Section 3.6 works.

4.1. Limits of classes of databases

Let $\mathcal{C}$ be a class of structures over a schema Σ. We allow $\mathcal{C}$ to contain both finite or infinite structures. By $\mathrm{substructures}_{\mathrm{fin}}(\mathcal{C})$ we denote the class of all *finite* structures which embed into some structure in $\mathcal{C}$.

LEMMA 4.1. *Let $\mathcal{C}$ and $\mathcal{D}$ be two classes of structures over a schema Σ, such that*

$$\text{substructures}_{\text{fin}}(\mathcal{C}) = \text{substructures}_{\text{fin}}(\mathcal{D}).$$

Then the the following conditions are equivalent for a database-driven system:

- *There is an accepting run driven by some structure from $\mathcal{C}$;*
- *There is an accepting run driven by some structure from $\mathcal{D}$.*

PROOF. Consider a database-driven system $\mathcal{A}$ over the schema Σ. The proof follows from the immediate fact that if $\mathbb{A}$ is a substructure of $\mathbb{B}$, then any run ρ driven by $\mathbb{A}$ induces a run of ρ' of driven by $\mathbb{B}$, and conversely, any run ρ' driven by $\mathbb{B}$ which only uses values from $\mathbb{A}$ induces a run ρ driven by $\mathbb{A}$. This is where we need the assumption that for infinite runs we find only finitely many data values. □

The lemma shows that from the point of view of database-driven systems, instead of considering a class $\mathcal{C}$, we may consider the class $\text{substructures}_{\text{fin}}(\mathcal{C})$, which is often better behaved than $\mathcal{C}$. Because of this, we often assume that the class $\mathcal{C}$ is closed under substructures. The lemma also allows us to go in the other direction – to consider a *single* structure $\mathbb{C}$ instead of the class $\mathcal{C}$, as we now describe. If $\mathcal{C}$ is a class of finite structures, then we say that a structure $\mathbb{C}$ is a *limit* of the class $\mathcal{C}$ if up to isomorphism, the substructures of $\mathbb{C}$ are precisely the structures in $\mathcal{C}$. As a consequence of this definition, $\text{substructures}_{\text{fin}}(\mathcal{C}) = \text{substructures}_{\text{fin}}(\{\mathbb{C}\})$. This, together with Lemma 4.1, yields the following.

FACT 4.2. *Let $\mathcal{C}$ be a class of finite structures over a common schema having a limit $\mathbb{C}$. Then the following conditions are equivalent for a database-driven system:*

- *There is an accepting run driven by some finite structure from $\mathcal{C}$;*
- *There is an accepting run driven by $\mathbb{C}$.*

Example 4.3. Suppose that the schema is empty (has only equality), and we are interested in the class $\mathcal{C}$ of all finite structures over this schema. Consider also the natural numbers – or any other countably infinite set – with equality. For a database-driven register automata, instead of asking if there is a finite run driven by some database from the class $\mathcal{C}$, we can ask if there is a finite run driven by the natural numbers with equality. The answers to both questions turn out to be the same, because the natural numbers with equality are a limit of the class $\mathcal{C}$.

Note that not every class $\mathcal{C}$ has a limit – a necessary condition for the existence of a limit of $\mathcal{C}$ is that $\mathcal{C}$ is closed under substructures. Also observe that for a given class $\mathcal{C}$, there might be several nonisomorphic limits. For instance, if $\mathcal{C}$ is the class of finite total orders, then any infinite total order can be used as the a limit structure, e.g. the natural numbers with order, the rational numbers with order, or a closed interval of rational numbers, with order. As we will see below, for algorithmic reasons, the rational numbers will be preferable as a limit for the class of finite total orders because the rational numbers are homogeneous, in the following sense.

4.2. Homogeneity

A (possibly infinite) structure is called *homogeneous* if every partial isomorphism between its finitely generated substructures extends to an automorphism of the entire structure.

Example 4.4. Consider the ordered integers $(\mathbb{Z}, \leq)$. The automorphisms of this structure are the translations, i.e. functions of the form $x \mapsto x+k$. This structure is not homogeneous, because the partial isomorphism which maps 0 to 0 and 1 to 2 does not extend to a full automorphism. For similar reasons, the natural numbers with order $(\mathbb{N}, \leq)$ are also not homogeneous, this structure even has no automorphisms. It is not difficult to see that,

in order to be homogeneous, a total order cannot have endpoints and must be dense. In particular, the real numbers and the rational numbers are homogeneous.

We have advocated that the existence of a homogeneous limit for $\mathcal{C}$ is a desirable property for manipulating database driven systems over $\mathcal{C}$. But such a limit does not always exist, even if $\mathcal{C}$ is closed under substructures.

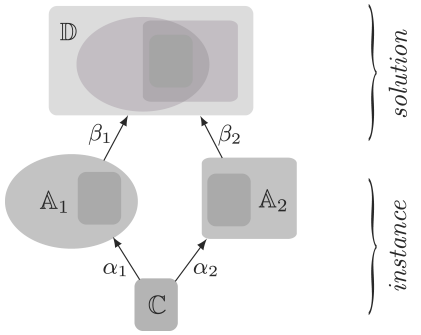
Example 4.5. Consider the class $\mathcal{C}$ of all graphs which are disjoint unions of paths. Suppose that $\mathbb{A}$ is a homogeneous limit of $\mathcal{C}$. Then $\mathbb{A}$ contains an induced path of length 3, and also an induced path of length 2. By homogeneity, there is an automorphism of $\mathbb{A}$ which maps the endpoints of the first path to the endpoints of the second path. It is easy to see that then $\mathbb{A}$ must contain a substructure which is isomorphic to a cycle, which is not in $\mathcal{C}$, a contradiction.

Classes of finite structures $\mathcal{C}$ which have a homogeneous limit have been characterized by Fraïssé in terms of several closure properties of $\mathcal{C}$, the most important one being closure under amalgamation, which we describe in the following section.

4.3. Amalgamation

An *instance of amalgamation* consists of two embeddings of the same database $\mathbb{C}$ into two other databases:

$$\alpha_1 : \mathbb{C} \to \mathbb{A}_1 \qquad \alpha_2 : \mathbb{C} \to \mathbb{A}_2.$$

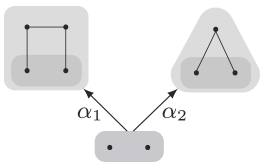


A *solution* to the instance is a database $\mathbb{D}$, together with embeddings

$$\beta_1 : \mathbb{A}_1 \to \mathbb{D} \qquad \beta_2 : \mathbb{A}_2 \to \mathbb{D}.$$

such that the diagram above commutes, i.e. $\beta_1 \circ \alpha_1 = \beta_2 \circ \alpha_2$.

Example 4.6. Consider a schema with one binary relation. The class of forests (understood as undirected graphs) is not closed under amalgamation: the instance depicted below does not have a solution which is a forest.



4.4. Fraïssé classes and their limits

A class of structures is called a *Fraïssé class* if all its structures are finitely generated, and:

- is *closed under amalgamation*, i.e. if every instance of amalgamation which uses structures from the class has a solution also in the class;
- is *closed under embeddings*, i.e., if it contains every finitely generated structure which embeds into some structure from the class;
- has the *joint embedding property*, which means that every two structures from the class can be embedded into a single structure from the class.

The definition above talks about finitely generated structures, because this is the classical definition of a Fraïssé class (see [Hodges 1997, Chapter 6]). However, in this paper we are only interested in the special case where the structures are actually finite.

Remark 4.7. The joint embedding property follows from closure under amalgamation and embeddings if there exists a finitely generated structure which embeds into any other structure in the class. This is the case when the schema has no constant symbols (nullary function symbols), since then the empty structure embeds into any structure in the class.

In general, however, the joint embedding property does not follow from amalgamation. For instance, consider a schema which consists of two constants c_1 and c_2, i.e. functions of arity zero. Consider the class of all finite structures over this schema. This class is closed under amalgamation and embedded substructures. However, a structure where the two constants are equal, and a structure where the two constants are different, cannot be jointly embedded into a single structure.

Example 4.8. Fraïssé classes over a schema with one binary relation symbol include: all finite linear orders, all finite undirected graphs, all finite directed graphs, and all equivalence relations.

Instead of proving that a class $\mathcal{C}$ is closed under amalgamation, it is enough to prove that it is closed under *inclusion amalgamation*. An instance of inclusion amalgamation consists of two structures $\mathbb{A}$ and $\mathbb{B}$ that are *consistent*, i.e. the functions and predicates are defined the same way on the elements that appear in both domains. A solution of inclusion amalgamation is a database $\mathbb{C}$ that contains both $\mathbb{A}$ and $\mathbb{B}$ as substructures.

LEMMA 4.9. *Let $\mathcal{C}$ be a class of structures closed under isomorphism. Then $\mathcal{C}$ is closed under amalgamation if and only if it is closed under inclusion amalgamation.*

An important property of a Fraïssé classes is that every Fraïssé class has a homogeneous limit structure.

THEOREM 4.10 (FRAÏSSÉ, SEE [HODGES 1997, THEOREM 6.1.2]). *Let $\mathcal{C}$ be a class of finitely generated structures. Then $\mathcal{C}$ has a countable homogeneous limit if and only if $\mathcal{C}$ is Fraïssé.*

The limit from the theorem is unique up to isomorphism, and is called the *Fraïssé limit* of the class $\mathcal{C}$.

Note that in our case $\mathcal{C}$ will be a class of finite structures, hence finitely generated.

Example 4.11. The Fraïssé limit of the class of all finite total orders is the unique dense total order without endpoints, namely the rational numbers. The Fraïssé limit of the class of all finite sets (seen as finite structures with only equality) is any countable structure with only equality, say the natural numbers. The Fraïssé limit of the class of all undirected graphs is the Rado graph, also called the infinite random graph.

5. EVALUATING FIXPOINT LOGICS OVER HOMOGENEOUS STRUCTURES

As described in Section 3.6, to solve the emptiness problem for database-driven systems, we will reduce it to evaluating formulas of transitive closure logic and least fixpoint logic in configuration structures. In this section we recall the syntax and semantics of these logics, and show how they can be evaluated in homogeneous structures.

5.1. Syntax and semantics of transitive closure and least fixpoint logics

In this section we describe the two logics that will be used to solve the emptiness problem for database-driven systems, namely transitive closure logic and least fixpoint logic. In terms of expressive power, transitive closure logic is a fragment of least fixpoint logic, but we use transitive closure logic when possible because its evaluation has better complexity.

Transitive closure logic. Transitive closure logic is obtained from first-order logic by the following operator, which can be used in a nested way. Let $\bar{x}, \bar{y}, \bar{u}, \bar{v}$ be tuples of variables, not necessarily disjoint, all of the same length n. Suppose that φ is a formula of transitive closure logic with free variables included in $\bar{x}\bar{y}$. Then

$$(TC_{\bar{x}\bar{y}}.\ \varphi)(\bar{u}\bar{v})$$

is a formula of transitive closure logic, with free variables $\bar{u}\bar{v}$. The semantics of the formula is the transitive closure of the semantics of φ, seen as a binary relation on n-tuples. Note that the TC operator needs to be annotated by an ordering of the variables, to indicate which variables are in the source and which variables are in the target of the relation whose transitive closure is used.

Example 5.1. Whether a database-driven register automaton $\mathcal{A}$ has an accepting run driven by $\mathbb{C}$ can be expressed as a formula of transitive closure logic over the configuration structure $\mathrm{Conf}(\mathbb{C}, \mathcal{A})$, namely

$$\exists u \exists v.\ I(u) \wedge F(v) \wedge (TC_{xy}.\ \Delta(x, y))(u, v)$$

where Δ is the name of the binary relation of $\mathrm{Conf}(\mathbb{C}, \mathcal{A})$, I the subset of initial configurations and F the subset of accepting configurations. The formula simply says that an accepting configuration is reachable from some initial configuration via the transitive closure of the transition relation.

Least fixpoint logic. To describe nonemptiness for branching models, such as database-driven tree automata, transitive closure logic is no longer expressive enough, and least fixpoint logic will be needed.

Least fixpoint logic is obtained from first-order logic by the following operator, which can be used in a nested way. Let φ be a formula of least fixpoint logic over a schema Σ, and let $R \in \Sigma$ be a predicate such that every appearance of R in φ is under an even number of negations. Let n be the arity of R and assume that φ has n free variables (we could allow parameters but we don't for simplicity of the presentation). Then

$$(\mu_{\bar{x}} R.\ \varphi)\ (\bar{y}) \tag{4}$$

is a formula of least fixpoint logic over the schema $\Sigma - \{R\}$, which has free variables $\bar{y}$. The semantics of the formula is defined as follows. In a given structure, φ yields a function from sets of n-tuples to sets of n-tuples, namely the function which inputs an interpretation of the predicate R, and outputs the set of valuations of the free variables for which it is true. By positivity, the function is monotone with respect to inclusion, and therefore admits a least fixpoint. The semantics of (4) is defined to be this least fixpoint.

Example 5.2. The transitive closure operator can be expressed in least fixpoint logic. The formula $(TC_{\bar{x}\bar{y}}.\ \varphi)(\bar{u}\bar{v})$ is equivalent to

$$\big(\mu_{\bar{x}\bar{y}}R.\ \varphi(\bar{x},\bar{y}) \vee \exists \bar{z} R(\bar{x},\bar{z}) \wedge R(\bar{z},\bar{y})\big)(\bar{u}\bar{v})$$

Example 5.3. Let $\mathcal{A}$ be a database-driven tree automaton, and $\mathbb{C}$ an input structure. The configuration structure $\mathrm{Conf}(\mathbb{C},\mathcal{A})$ is defined the same way as for database-driven register automata, except that the transition relation Δ is not binary but ternary: the first argument corresponds to the parent, and the other two arguments correspond to the two children. Let us adopt the convention that the initial states are those in the leaves, and the accepting state is in the root. The following formula of fixpoint logic, which has one fixpoint variable x, defines the set of configurations reachable by a run which has only initial states in the leaves:

$$(\mu_x R.\ I(x) \ \vee\ \exists y \exists z\ R(y) \wedge R(z) \wedge \Delta(x,y,z)).$$

Therefore, nonemptiness can be expressed by a formula of least fixpoint logic, which asks if the above fixpoint selects at least one accepting state.

5.2. Evaluation in homogeneous structures

In this section we describe the complexity of evaluating transitive closure logic and least fixpoint logic in a homogeneous structure $\mathbb{C}$. To analyze the complexity of this problem, we need to assume some good computational properties of the underlying structure $\mathbb{C}$. We start with defining these properties.

Complexity of embeddability. Let $\mathbb{C}$ be a homogeneous structure. *Embeddability* in $\mathbb{C}$ is the following decision problem: given on input a finite structure $\mathbb{A}$, decide whether $\mathbb{A}$ embeds into $\mathbb{C}$. If $\mathbb{C}$ is the Fraïssé limit of a class $\mathcal{C}$ of finite structures, then the embeddability problem is equivalent to membership in $\mathcal{C}$. The homogeneous structures that where mentioned earlier typically have embeddability decidable in PTime. For the following results, it will suffice that $\mathbb{C}$ has embeddability decidable in PSpace.

Blowup functions. The *blowup function* of a structure $\mathbb{C}$ is the function

$$\mathrm{blowup}_{\mathbb{C}} : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

that maps $n \in \mathbb{N}$ to the least upper bound on the size of n-generated substructures of $\mathbb{C}$. Note that in the absence of function symbols in the schema, every n-generated structure has size n and therefore $\mathrm{blowup}_{\mathbb{C}}(n) = n$.

If $\mathbb{C}$ is the Fraïssé limit of a class $\mathcal{C}$ of finite structures, then $\mathrm{blowup}_{\mathbb{C}}(n)$ is equal to the least upper bound on the size of n-generated structures in $\mathcal{C}$.

Example 5.4. Consider the structure $\mathbb{C}$, consisting of the set of integers equipped with a *bounded successor*, which is a binary function f defined by

$$f(x,y) = \begin{cases} z & \text{if } z \text{ is the successor of } x \text{ and } x < z \leq y \\ x & \text{otherwise.} \end{cases}$$

This structure is homogeneous, and has embeddability decidable in PTime. However, the size of 2-generated substructures can be arbitrarily large, and therefore

$$\mathrm{blowup}_{\mathbb{C}}(2) = \infty.$$

Over this structure, emptiness of database-driven systems is undecidable. The idea is that a system with four registers can simulate finite runs of two-counter machines. One of the registers is used to store zero, and is unchanged during the run. One of the registers is used to store the maximal counter value that will appear during the run. The remaining two registers are used for the actual counters.

Let Σ be a schema. For $n \in \mathbb{N}$, let $s_\Sigma(n)$ denote the maximal size of a the standard description of a structure over schema Σ with domain $\{1, \ldots, n\}$, i.e., $s_\Sigma(n)$ is polynomial in $|\Sigma|$ and n, and exponential in the maximal arity of a symbol in Σ.

THEOREM 5.5. *Let $\mathbb{C}$ be a homogeneous structure over schema Σ, with embeddability decidable in* DSPACE(e) *and blowup function f such that $f(n)$ is finite for all n. The model checking problem $\mathbb{C} \models \varphi$ can be decided in time*

$$poly(|\varphi|) \cdot exp(e(s_\Sigma(f(l)))) \tag{5}$$

when φ is a sentence of fixpoint logic with l variables, and in space

$$poly(|\varphi|) \cdot poly(e(s_\Sigma(f(l)))) \tag{6}$$

when φ is a sentence of transitive closure logic with l variables.

The rest of Section 5 is devoted to showing the above theorem.

Fix a structure $\mathbb{C}$ as in the statement of the theorem. If $\bar{x}$ is a tuple of variables, then by $\bar{x}$-valuation we mean a valuation of the variables in $\bar{x}$ by elements of $\mathbb{C}$. Define the *type* of an $\bar{x}$-valuation to be the isomorphism type of the structure generated by its image, with special constants for each variable in $\bar{x}$. In other words, two $\bar{x}$-valuations have the same type if there is a partial automorphism π of $\mathbb{C}$ which maps one valuation to the other, i.e. makes the following diagram commute

$$\begin{array}{ccc} & \bar{x} & \\ {\scriptstyle \eta_1}\swarrow & & \searrow{\scriptstyle \eta_2} \\ \mathbb{C} & \xrightarrow{\quad\pi\quad} & \mathbb{C} \end{array} \tag{7}$$

If $|\bar{x}|$ denotes the number of variables in $\bar{x}$, then each $\bar{x}$-valuation induces a structure of size $f(|\bar{x}|)$. The number of isomorphism types of such structures is exponential in $f(|\bar{x}|)$ and each of them can be represented in space $s_\Sigma(f(|\bar{x}|))$. An isomorphism type of structures over the schema of $\mathbb{C}$ is called a $\bar{x}$-type if it is the type of some $\bar{x}$-valuation. By the assumption that embeddability is decidable in DSPACE(e), deciding if an isomorphism type is a $\bar{x}$-type can be done in space $e(s_\Sigma(f(|\bar{x}|)))$. In particular any algorithm trying naively all $\bar{x}$-types will take time $exp(e(s_\Sigma(f(l))))$.

Since $\mathbb{C}$ is homogeneous, every partial automorphism extends to a full automorphism, and therefore two $\bar{x}$-valuations have the same type if and only if there is a full automorphism π such that the diagram in (7) commutes. Both for transitive closure logic and least fixpoint logic, the truth value of logical formulas is preserved under automorphisms, and therefore $\bar{x}$-valuations of the same type satisfy the same formulas of these logics.

Let φ be a formula of least fixpoint logic or transitive closure logic, with free variables $\bar{x}$. Define $[\varphi]_\mathbb{C}$ to be the set of types of $\bar{x}$-valuations that satisfy φ; this is a subset of all $\bar{x}$-types. By the discussion in the previous paragraph, whether or not a $\bar{x}$-valuation satisfies φ depends only on its type. By induction on the structure of φ, we will show that $[\varphi]_\mathbb{C}$ can be computed in time and space as described in the statement of the lemma. The proof is essentially a naive evaluation algorithm except that the computation is done at the level of types instead of the elements of the domain.

First order constructs. The base case for atomic formulas, as well as conjunction, disjunction and negation, are done in the obvious way. For existential quantification, we use the projection operation from $x\bar{y}$-types to $\bar{y}$-types, which maps the type of a $x\bar{y}$-valuation to the type of the same valuation with variable x removed.

Fixpoint. Consider the least fixpoint operator $\mu_{\bar{x}} R.\psi$. Define a sequence $\psi_0, \psi_1, \ldots$ of formulas with free variables $\bar{x}$ as follows. Let $\psi_0 = \bot$. For $k \geq 0$, let ψ_{k+1} be the formula ψ in which each predicate $R(\bar{u})$ is replaced by $\psi_k(\bar{u})$. Assuming that $[\psi_k]_\mathbb{A}$ has already been

computed, then by induction assumption we can compute $[\psi_{k+1}]_{\mathbb{A}}$ in time

$$poly(|\psi|) \cdot exp(e(s_\Sigma(f(l)))),$$

where l is the number of (possibly bound) variables in ψ. Note that when going from $[\psi_k]_{\mathbb{A}}$ to $[\psi_{k+1}]_{\mathbb{A}}$, we store the entire set $[\psi_k]_{\mathbb{A}}$, which is the reason why the algorithm does not run in polynomial space. Because the predicate R occurs only positively in ψ, it follows that

$$[\psi_0]_{\mathbb{A}} \subseteq [\psi_1]_{\mathbb{A}} \subseteq [\psi_2]_{\mathbb{A}} \subseteq \dots$$

Since each element of the sequence is a subset of all $\bar{x}$-types, the sequence stabilizes in a number of steps that is bounded by the number of $\bar{x}$-types, which is at most exponential in $f(|\bar{x}|)$. The value at which the sequence stabilizes is the least fixpoint.

Transitive closure. Consider the transitive closure $TC_{\bar{x}\bar{y}}\psi$. Let n be the number of variables in $\bar{x}$, which is the same as the number of variables in $\bar{y}$. Let $\bar{a}$ and $\bar{b}$ be two n-tuples of elements in $\mathbb{C}$. We say that $\bar{a}$ and $\bar{b}$ are *ψ-connected* if the formula ψ is satisfied by the valuation which uses $\bar{a}$ for $\bar{x}$ and which uses $\bar{b}$ for $\bar{y}$. A *ψ-path* is a finite sequence of n-tuples, where each two consecutive tuples are ψ-connected. By definition of the transitive closure operator, a valuation $\bar{a}\bar{b}$ satisfies $TC_{\bar{x}\bar{y}}\psi$ if and only if there is a ψ-path from $\bar{a}$ to $\bar{b}$. Consider the directed graph where vertices are $\bar{x}\bar{y}$-types, and there is an edge from τ to τ' if there is an n-tuple $\bar{a}$ and ψ-connected n-tuples $\bar{b}$ and $\bar{b}'$ such that

(1) τ is the type of the valuation which maps $\bar{x}$ to $\bar{a}$ and which maps $\bar{y}$ to $\bar{b}$;
(2) τ' is the type of the valuation which maps $\bar{x}$ to $\bar{a}$ and which maps $\bar{y}$ to $\bar{b}'$.

The graph characterizes the transitive closure thanks to the following lemma, which is shown by induction on the number of steps in the transitive closure.

LEMMA 5.6. *An $\bar{x}\bar{y}$-type belongs to $[TC_{\bar{x}\bar{y}}\psi]_{\mathbb{A}}$ if and only if it can be reached in the graph above from some vertex which corresponds to a $\bar{x}\bar{y}$-valuation satisfying $\bar{x} = \bar{y}$.*

The vertices of the graph, which are $\bar{x}\bar{y}$-types, can be represented in space polynomial in $f(\bar{x}\bar{y})$. Thanks to the induction assumption, the edges of the graph can be computed in space as required by the lemma. Therefore, $[TC_{\bar{x}\bar{y}}\psi]_{\mathbb{A}}$ can be computed by using Savitch's theorem to test reachability in the graph.

This completes the proof of Theorem 5.5.

6. COMPLEXITY OF EMPTINESS FOR DATABASE-DRIVEN SYSTEMS

In this section we combine the results from the previous sections to show that it is decidable whether a database-driven system has an accepting run driven by a database in a given Fraïssé class. We observe that each of our database-driven systems can be seen as a "definable" structure over the Fraïssé limit of the class and that the existence of an accepting run can be expressed by a sentence of fixpoint logic. We then show that it is decidable whether a sentence of fixpoint logic is true or not on a structure definable over a Fraïssé limit.

6.1. Definable sets and structures

We consider two operations that transform structures into other structures, namely copying and interpretation.

Copying. Let $\mathbb{A}$ be a structure with domain A. For $k \in \mathbb{N}$, define $k \times \mathbb{A}$ to be the following structure. The domain is $\{1, \dots, k\} \times A$, and the schema is the same as in $\mathbb{A}$, except that for every $i \in \{1, \dots, k\}$ there is a unary function $copy_i$ defined by $(j, a) \mapsto (i, a)$. The symbols from the original schema are defined as follows, for relations and functions respectively:

$$\begin{array}{lll} R^{k\times\mathbb{A}}((i_1,a_1),\dots,(i_k,a_k)) & \text{iff} & R^{\mathbb{A}}(a_1,\dots,a_k) \\ f^{k\times\mathbb{A}}((i_1,a_1),\dots,(i_k,a_k)) & = & (1, f^{\mathbb{A}}(a_1,\dots,a_k)) \end{array}$$

The following simple lemma follows immediately from the definitions.

LEMMA 6.1. *Assume that $\mathbb{A}$ is homogeneous, has embeddability in* DSPACE(e)*, and has blowup function f. Then $k \times \mathbb{A}$ is homogeneous, has embeddability in* DSPACE(e)*, and blowup function $n \mapsto k \cdot f(n)$.*

Interpretation. Consider two schemas Γ and Σ, called the source schema and target schema. For $n \in \mathbb{N}$, an *n-dimensional interpretation* between these schemas is a family of formulas $\{\varphi_\sigma\}_{\sigma \in \Sigma}$ over the source schema, such that when σ is a k-ary relation then φ_σ has arity $n \cdot k$, and when σ is a k-ary function then φ_σ has arity $n \cdot (k+1)$. For a structure $\mathbb{A}$ over the source schema, we define the output of the interpretation to be the structure over the target schema where the domain is n-tuples of the domain of $\mathbb{A}$, and the functions and relations are defined in the natural way. The output might be undefined if the formulas for functions do not define functions, i.e. the output is defined if and only if for every k-ary function symbol f, the interpretation of φ_f in $\mathbb{A}$ is a function form the first $k \cdot n$ arguments to the last n arguments.

In model theory, a more general notion of interpretation is sometimes used, where the universe of the output is n-tuples modulo some partial equivalence relation; a formula defining this partial equivalence relation is also part of the interpretation. The results below would also hold for the more general notion, with the same proofs.

The following lemma establishes the complexity of evaluating fixpoint logics on structures obtained from a homogeneous structure by first copying and then an interpretation. Recall that $s_\Sigma(n)$ denotes the maximal size of a description of a structure over schema Σ with domain $\{1, \ldots, n\}$.

LEMMA 6.2. *Let $\mathbb{A}$ be a homogeneous structure over schema Σ, with embeddability decidable in* DSPACE(e) *and blowup function f, and let φ be a sentence of least fixpoint logic using l variables. For every k and every n-dimensional interpretation I,*

$$I(k \times \mathbb{A}) \models \varphi$$

can be decided in time

$$poly(|\varphi|, |I|) \cdot exp(k, e(s_\Sigma(f(l \cdot n)))). \tag{8}$$

If φ is a formula of transitive closure logic, then evaluation can be done in space

$$poly(|\varphi|, |I|) \cdot poly(k, e(s_\Sigma(f(l \cdot n)))). \tag{9}$$

PROOF. By substituting formulas, one can compute a formula $I(\varphi)$ such that

$$I(\mathbb{B}) \models \varphi \qquad \text{iff} \qquad \mathbb{B} \models I(\varphi).$$

holds for every structure $\mathbb{B}$, in particular for $\mathbb{B} = k \times \mathbb{A}$. The formula $I(\varphi)$ and the time to compute it are polynomial in $|I|$ and $|\varphi|$, and it uses $l \cdot n$ variables. The result then follows by applying Theorem 5.5, whose assumptions are met thanks to Lemma 6.1, and the fact that $s_{\Sigma_k}(n) = poly(k, s_\Sigma)$, where Σ_k is the schema of $k \times \mathbb{A}$. □

As an immediate corollary of Lemma 6.2, we get upper bounds on the complexities of nonemptiness for various automata models.

COROLLARY 6.3. *Let $\mathcal{C}$ be a Fraïssé class over schema Σ of maximal arity r, with blowup function f and membership in* DSPACE(e)*. Then, over $\mathcal{C}$,*

(1) *Emptiness for a database-driven register automaton $\mathcal{A}$ using n registers can be decided in space $poly(|\Sigma| \cdot e(f(2n)^r) \cdot \text{size}(\mathcal{A}))$.*
(2) *Emptiness for a database-driven Büchi register automaton $\mathcal{A}$ using n registers can be decided in space $poly(|\Sigma| \cdot e(f(2n))^r \cdot \text{size}(\mathcal{A}))$.*

(3) *Emptiness for a database-driven register tree automaton using n registers can be decided in time* $exp(|\Sigma| \cdot e(f(3n))^r) \cdot poly(\text{size}(\mathcal{A}))$.
(4) *Emptiness for a database-driven pushdown register automaton $\mathcal{A}$, where n is the maximum of control and stack registers, can be decided in time* $exp(|\Sigma| \cdot e((3n))^r) \cdot poly(\text{size}(\mathcal{A}))$.

PROOF. Let $\mathbb{C}$ be the homogeneous Fraïssé limit of $\mathcal{C}$ as given by Theorem 4.10.

(1) We start with a database-driven register automaton $\mathcal{A}$. By Fact 4.2 it is enough to test for the existence of an accepting run driven by $\mathbb{C}$. As explained in Example 5.1, this boils down to evaluating a formula φ of transitive closure in the configuration structure $\text{Conf}(\mathbb{C}, \mathcal{A})$. It is not difficult to see that this configuration structure is equal to $I(k \times \mathbb{C})$ where k is the number of states of $\mathcal{A}$ and I is an interpretation of dimension n, where n is the number of registers of $\mathcal{A}$, which can be obtained from the transition relation of $\mathcal{A}$ in polynomial time. Therefore, emptiness is equivalent to evaluating the formula φ in $I(k \times \mathbb{C})$. Since φ – the formula that tests reachability of some final state from the initial state – has 2 variables, it follows from Lemma 6.2 that the emptiness problem can be solved in space as in the statement of the corollary (recall that $s_\Sigma(n)$ is polynomial in $|\Sigma|$ and n and exponential in r).
(2) Consider now the case of Büchi register automata. The proof is the same as in the previous case, except that in the transition structure $\text{Conf}(\mathbb{C}, \mathcal{A})$ we evaluate a slightly different formula of transitive closure logic. As we have commented in Section 3.2, a Büchi automaton is nonempty if and only if for some database in $\mathcal{C}$, it admits a run that visits some accepting configuration infinitely often. This is the same as saying that over the limit $\mathbb{C}$, there exists: an initial configuration x and an accepting configuration y, such that y is reachable from both x and itself. This can be expressed by a formula of transitive closure logic with 2 variables, and hence we get the same complexity as in the previous case.
(3) For tree automata, we consider the transition structure and the least fixpoint logic formula as defined in Example 5.3
(4) The proof for pushdown automata is a straightforward adaptation of the standard algorithm for pushdown emptiness, which happens to be the same construction as when converting a pushdown automaton into a context-free grammar. Consider a pushdown automaton $\mathcal{A}$. Define the transition structure $\text{Conf}(\mathbb{C}, \mathcal{A})$ to be the structure where the domain is the disjoint union of the sets
   C pairs consisting of a control state and a valuation of the registers;
   S pairs consisting of a stack label and a valuation of the stack registers.
   The schema has the following relations: unary relations I, F describing the subsets of C that use initial and final control states respectively, and ternary relations $pop, push$ describing subsets of $C \times S \times C$. Define a *well-nested* run of the pushdown automaton to be a run where the stack is the same at the beginning and the end, and none of the contents of this stack are popped throughout the run, although additional tokens may be pushed and then popped. Define the *profile* of such a run to be the triple in $C \times S \times C$ which consists of the: control state and valuation of the control registers at the beginning of the run, the stack label and valuation of the stack registers in the topmost stack token, and the control state and valuation of the control registers at the end of the run. It is easy to see that the profiles of well-nested runs is the least set R of triples that satisfies both formulas below:

$$\begin{aligned} \forall x_1, x_2, x_3, y \quad & R(x_1, y, x_2) \wedge R(x_2, y, x_3) \Rightarrow R(x_1, y, x_3) \\ \forall x_1, x_2, x_3, x_4, y, z \quad & push(x_1, y, x_2) \wedge R(x_2, y, x_3) \wedge pop(x_3, y, x_4) \Rightarrow R(x_1, z, x_4) \end{aligned}$$

A pushdown automaton is nonempty if and only if some well-nested run has a profile that has an initial state on the first coordinate and a final state on the last coordinate, which is a property that can be defined in least fixpoint logic.

□

The bounds from the above corollary are tight in the following sense. Assume that the structure $\mathbb{A}$ has a domain of size at least two. As shown in Lemma 3.2, the emptiness problem is PSPACE-hard for any Fraïssé limit which has a domain of size at least two, already for finite words. For pushdown automata, the emptiness for pushdown register automata is EXPTIME-hard. The proof is the same as in Lemma 3.2, only now we can simulate an alternating machine with space n. The configurations of the alternating machine can be stored on the stack of the pushdown automaton.

7. SEMI-FRAÏSSÉ CLASSES

For some of the classes we are interested in, Theorem 5.5 will not work, since the class is not Fraïssé. However, sometimes, the class is closely related to a Fraïssé class, so that Corollary 6.3 can still be applied. In this section we study two ways of modifying a class $\mathcal{C}$ to obtain a Fraïssé class:

- We show that if $\mathcal{C}$ is closed under amalgamation and under embeddings, but does not have the joint embedding property, then it can be converted into a Fraïssé class $\mathcal{C}^*$, and the conclusion of Corollary 6.3 still holds.
- We show that if $\mathcal{C}$ can be obtained as a *reduct* of a Fraïssé class $\mathcal{D}$, then the conclusion of Corollary 6.3 still holds.

7.1. Classes closed under amalgamation and embeddings

We show that if a class $\mathcal{C}$ of finite databases is closed under amalgamation and embeddings, but does not have the joint embedding property, then we still can deduce the complexity of the emptiness problem of database-driven systems over $\mathcal{C}$.

Fix a schema Σ. A *type* τ is a 0-generated structure (this corresponds to quantifier-free 0-types). If $\mathcal{C}$ is a class of finite databases, then by types($\mathcal{C}$) we denote the types in substructures$_{\text{fin}}(\mathcal{C})$. If Σ contains no constant symbols, then types($\mathcal{C}$) consists of only one structure, namely the empty structure. In general, types($\mathcal{C}$) contains the substructures of structures in $\mathcal{C}$ which are generated by the constants.

If $\tau \in$ types($\mathcal{C}$) and $\mathbb{C}$ is a structure such that τ embeds into $\mathbb{C}$, then we say that $\mathbb{C}$ *has type* τ. By $\mathcal{C}_\tau$ we denote the set of all structures in $\mathcal{C}$ which have type τ. Since every structure has some type, it follows that

$$\mathcal{C} = \bigcup_{\tau \in \text{types}(\mathcal{C})} \mathcal{C}_\tau. \tag{10}$$

LEMMA 7.1. *Let $\mathcal{C}$ be a class of finite databases over a common schema Σ which is closed under amalgamation and under embeddings, and has blowup function f and membership in* PSPACE. *For each $\tau \in$ types($\mathcal{C}$), $\mathcal{C}_\tau$ is a Fraïssé class with blowup function bounded by f and membership in* PSPACE.

PROOF. It is easy to see that $\mathcal{C}_\tau$ is closed under embeddings and under amalgamation. To see that $\mathcal{C}_\tau$ has the joint embedding property, observe that if $\mathbb{A}_1, \mathbb{A}_2$ are two structures in $\mathcal{C}_\tau$, then τ embeds into both $\mathbb{A}_1$ and $\mathbb{A}_2$. This defines an amalgamation instance, whose solution yields a structure $\mathbb{D}$ which jointly embeds $\mathbb{A}_1$ and $\mathbb{A}_2$. Therefore, $\mathcal{C}_\tau$ is a Fraïssé class. Since $\mathcal{C}_\tau \subseteq \mathcal{C}$, the blowup function of $\mathcal{C}_\tau$ is bounded by f. To test whether a structure $\mathbb{C}$ belongs to $\mathcal{C}_\tau$, it suffices to test whether it belongs to $\mathcal{C}$ and whether its 0-generated substructure is isomorphic to τ, which can be done in PSPACE. □

COROLLARY 7.2. *Let $\mathcal{C}$ be a class which is closed under amalgamation and under embeddings, with blowup function f and membership in* PSPACE. *Then emptiness of database-driven systems over $\mathcal{C}$ can be decided with complexity as described in Corollary 6.3.*

PROOF. Consider a database-driven system over $\mathcal{C}$. By the equality (10), there is an accepting run driven by $\mathbb{C} \in \mathcal{C}$ if and only if there is an accepting run driven by $\mathbb{C} \in \mathcal{C}_\tau$, for some $\tau \in \text{types}(\mathcal{C})$.

Each type $\tau \in \text{types}(\mathcal{C})$ has cardinality at most $f(0)$, since it is a 0-generated structure. Testing whether a structure τ of cardinality at most $f(0)$ is a type of $\mathcal{C}$ can be done in polynomial space with respect to $f(0)$, and so the set $\text{types}(\mathcal{C})$ can be iterated the same space. For each type τ in this set, emptiness of $\mathcal{S}$ over $\mathcal{C}_\tau$ can be done as described in Corollary 6.3, since by Lemma 7.1 $\mathcal{C}_\tau$ is a Fraïssé class with blowup function at most f and membership in PSPACE. □

7.2. Semi-Fraïssé classes

We show that when a class $\mathcal{C}$ is not even closed under amalgamation, then sometimes it can be enhanced to a Fraïssé class so that decidability of database-driven systems can be deduced. Consider the following example.

Example 7.3. Consider graphs that are 2-colorable or, equivalently, have no odd-length cycle. This class is $\text{HOM}(\mathbb{H})$ where $\mathbb{H}$ is a 2-clique, over the schema Σ consisting of one binary relation.

This class is not closed under amalgamation. Indeed, there is an odd-length cycle in every solution of the instance of amalgamation depicted in Example 4.6.

A solution to the problem is to consider not 2-colorable graphs, but 2-*colored* graphs, i.e. graphs with a 2-coloring. This corresponds to considering an extended schema Γ, with the original binary edge relation, and two unary predicates denoting the colors. The 2-clique $\mathbb{H}$ lifts to a canonical graph $\tilde{\mathbb{H}}$ over this schema, where each node gets a different color. The class $\text{HOM}(\tilde{\mathbb{H}})$ is now closed under amalgamation, and is Fraïssé.

In the example above, we added some structure to the databases in order to recover amalgamation. We formalize this strategy below. Let $\mathbb{G}$ be a database over a schema Γ and let Σ be a subset of the schema Γ. The Σ-*reduct* of $\mathbb{G}$, denoted $\Sigma(\mathbb{G})$, is the set domain of $\mathbb{G}$, together with the relations in Σ. The Σ-reduct of a class $\mathcal{D}$ of databases over Γ, denoted by $\Sigma(\mathcal{D})$, is defined pointwise.

The following lemma is immediate.

LEMMA 7.4. *Let $\mathcal{D}$ be a class of finite databases over a schema Γ. If $\mathcal{C}$ is a class of finite databases over a schema $\Sigma \subseteq \Gamma$ such that*

$$\text{substructures}_{\text{fin}}(\mathcal{C}) = \text{substructures}_{\text{fin}}(\Sigma(\mathcal{D})), \tag{11}$$

then the following conditions are equivalent for a database-driven system $\mathcal{S}$ over the schema Σ:

- *$\mathcal{S}$ has an accepting run driven by some database in $\mathcal{C}$,*
- *$\mathcal{S}$, treated as a system over the schema $\mathcal{D}$, has an accepting run driven by some database in $\mathcal{D}$.*

We call a class $\mathcal{C}$ *semi-Fraïssé* if there exists a class $\mathcal{D}$ which is closed under amalgamations and embeddings and such that equation (11) holds.

Consider a semi-Fraïssé class $\mathcal{C}$. From Lemma 7.4 it follows that the decidability of the emptiness problem over $\mathcal{C}$ reduces to the decidability of the emptiness problem over $\mathcal{D}$. By Corollaries 7.2 and 6.3 the complexity of the latter does not depend on the automaton depends but only on the function $\text{blowup}_{\mathcal{D}}$, the embeddability function e and the schema Σ.

Hence it is crucial to witness semi-Fraïssé using a class of structures with these parameters small.

7.3. Data values

In this section we show closure properties over Fraïssé classes. These closure properties will be important to show our main results. As a main consequence they show that we can treat separately the structure of the database (for instance XML trees) and the structure of the data values they manipulate (for instance natural number with order). The method is again very general – the data values themselves may carry some structure; we only require that the data values come from a homogeneous relational structure. After introducing some preliminary notions, we describe this general setting.

Data values. Fix a homogeneous structure $\mathbb{F}$, whose elements will model *data values*. We assume that the schema of $\mathbb{F}$ is purely relational, i.e. does not contain function symbols. For instance $\mathbb{F}$ could be the structure $\langle \mathbb{N}, \sim \rangle$, where $\sim$ is the equality relation, or $\langle \mathbb{Q}, < \rangle$.

Consider a finite database $\mathbb{A}$ over a schema Σ. Let $\lambda : \mathbb{A} \to \mathbb{F}$ be any labeling of the elements of $\mathbb{A}$ by elements of $\mathbb{F}$. We denote by $\mathbb{A} \otimes \lambda$ the (finite) database extending $\mathbb{A}$ by symbols from the schema of $\mathbb{F}$, which are interpreted in $\mathbb{A} \otimes \lambda$ via the mapping λ: if R is a relation symbol in $\mathbb{F}$, then

$$(\mathbb{A} \otimes \lambda) \models R(x_1, \ldots, x_k) \iff \mathbb{F} \models R(\lambda(x_1), \ldots, \lambda(x_k)).$$

The schema of $\mathbb{A} \otimes \lambda$ is therefore the union of the schema of $\mathbb{A}$ and the schema of $\mathbb{F}$. The database $\mathbb{A} \otimes \lambda$ can be seen as a database whose elements are additionally labeled by data values, and the database contains relation symbols from $\mathbb{F}$ allowing to compare the data values. If $\mathcal{C}$ is a class of finite databases, then by $\mathcal{C} \otimes \mathbb{F}$ we denote the class of databases of the form $\mathbb{A} \otimes \lambda$, where $\mathbb{A} \in \mathcal{C}$ and λ is a mapping from $\mathbb{A}$ to $\mathbb{F}$. By $\mathcal{C} \odot \mathbb{F}$ we denote subset of $\mathcal{C} \otimes \mathbb{F}$ consisting of those databases $\mathbb{A} \otimes \lambda$, where the mapping λ is *injective*, i.e. each node gets a different data value. We consider the two variants $\otimes$ and $\odot$ because in relational databases, we want every value to be unique – to avoid redundancy – while in XML databases, attributes are used for identifying distinct nodes (see Examples 7.5 and 7.6 below).

Example 7.5. Let w be a finite word over a finite alphabet seen as a database whose domain is the set of positions in w and whose relations are the linear order among the positions together with a unary predicate per letter of the alphabet. Let $\lambda \colon w \to \mathbb{N}$ be a labeling of the positions of w by natural numbers. Then the database $w \otimes \lambda$ can be seen as a word equipped with data values (or *attributes*); two positions x, y store the same attribute if $x \sim y$. These structures are often denoted as *data words* in the literature.

Example 7.6. Let $\mathbb{G}$ be a finite graph and let $\lambda \colon \mathbb{G} \to \mathbb{N}$ be an injective labeling of $\mathbb{G}$ by natural numbers. Then the database $\mathbb{G} \odot \lambda$ can be seen as a graph whose nodes are natural numbers: because λ is injective, we can identify a node x with the number $\lambda(x)$. If $\mathcal{G}$ denotes the class of all graphs, then the structures in $\mathcal{G} \odot \langle \mathbb{N}, \sim \rangle$ can be interpreted as graphs on natural numbers. Similarly, the structures in $\mathcal{G} \odot \langle \mathbb{Q}, < \rangle$ can be interpreted as graphs on rational numbers; in particular, their nodes are linearly ordered.

Using the theorem of Fraïssé and a construction for combining two Fraïssé classes into one class, we can obtain the following proposition:

PROPOSITION 7.7. *Let $\mathbb{F}$ be a purely relational homogeneous structure, such that embeddability is in* PSPACE. *Then, for any Fraïssé class $\mathcal{C}$ (over any schema), the classes $\mathcal{C} \otimes \mathbb{F}$ and $\mathcal{C} \odot \mathbb{F}$ are Fraïssé classes, with the same blowup function as $\mathcal{C}$.*

PROOF. We prove the more general case of $\mathcal{C} \odot \mathbb{F}$. The case of $\mathcal{C} \otimes \mathbb{F}$ can be easily reduced to this case, by observing that for the homogeneous structure

$$\tilde{\mathbb{F}} = \mathbb{F} \times \langle \mathbb{N}, \sim \rangle,$$

the class $\mathcal{C} \otimes \mathbb{F}$ is equivalent to the class $\mathcal{C} \odot \tilde{\mathbb{F}}$.

Let $\mathcal{F}$ denote the class of finite substructures of the homogeneous structure $\mathbb{F}$. In particular, $\mathcal{F}$ is a class of finite databases over a purely relational schema Σ. Assuming that $\mathbb{F}$ is nonempty, $\mathcal{F}$ contains at least one structure with a nonempty domain (the case of Proposition 7.7 when $\mathbb{F}$ is empty is easy).

The following simple lemma uses the fact that $\mathbb{F}$ is purely relational.

LEMMA 7.8. *Consider an instance of inclusion amalgamation in* $\mathcal{F}$

$$\mathbb{C} \subseteq \mathbb{A}_1, \qquad \mathbb{C} \subseteq \mathbb{A}_2. \tag{12}$$

Then for any finite set D *such that* $dom(\mathbb{A}_1) \cup dom(\mathbb{A}_2) \subseteq D$, *the instance* (12) *has a solution with domain* D.

PROOF. Notice that on relational structures, any subset induces a substructure, so one can easily restrict a solution to the amalgamation instance (12) to one whose domain is equal to $\mathrm{dom}(\mathbb{A}_1) \cup \mathrm{dom}(\mathbb{A}_2)$. Further, one can extend any structure in $\mathcal{F}$ to a structure in $\mathcal{F}$ whose domain contains some additional elements. The result follows from these two observations. □

We now continue with the proof of Proposition 7.7. Let Σ denote the schema of the databases in the class $\mathcal{F}$; and let T denote the schema of the databases in the class $\mathcal{C}$. We assume that Σ and T are disjoint.

By $\mathcal{C} \odot \mathcal{F}$ we denote the class of structures over the union schema $\Sigma \cup T$, whose Σ-projection belongs to $\mathcal{F}$ and T-projection belongs to $\mathcal{C}$. It is easy to see that $\mathcal{C} \odot \mathcal{F}$ is the same class as $\mathcal{C} \odot \mathbb{F}$ (up to isomorphisms of the databases). We now prove that the class $\mathcal{C} \odot \mathcal{F}$ is a Fraïssé class, and compute its blowup function.

PROOF. It is obvious that $\mathcal{C} \odot \mathcal{F}$ is a class which is closed under embeddings (this does not require the assumption that Σ is a purely relational schema).

We prove that $\mathcal{C} \odot \mathcal{F}$ is closed under amalgamation. The joint embedding property can be proved similarly.

Consider the following instance of inclusion amalgamation:

$$\mathbb{C} \subseteq \mathbb{A}_1, \qquad \mathbb{C} \subseteq \mathbb{A}_2, \tag{13}$$

with $\mathbb{A}_1, \mathbb{A}_2, \mathbb{C} \in \mathcal{C} \odot \mathcal{F}$. This instance induces instances of inclusion amalgamation in $\mathcal{F}$ and in $\mathcal{C}$, respectively:

$$\Sigma(\mathbb{C}) \subseteq \Sigma(\mathbb{A}_1), \qquad \Sigma(\mathbb{C}) \subseteq \Sigma(\mathbb{A}_2) \tag{14}$$

$$T(\mathbb{C}) \subseteq T(\mathbb{A}_1), \qquad T(\mathbb{C}) \subseteq T(\mathbb{A}_2). \tag{15}$$

Since $\mathcal{C}$ is a Fraïssé class there exists a solution to the second instance (15),

$$T(\mathbb{A}_1) \subseteq \mathbb{D}_T, \qquad T(\mathbb{A}_2) \subseteq \mathbb{D}_T \tag{16}$$

We now apply Lemma 7.8 to the first instance (14), and to $D = \mathrm{dom}(\mathbb{D}_T)$, yielding a solution

$$\Sigma(\mathbb{A}_1) \subseteq \mathbb{D}_\Sigma, \qquad \Sigma(\mathbb{A}_1) \subseteq \mathbb{D}_\Sigma \tag{17}$$

in which the domain of $\mathbb{D}_\Sigma$ is equal to D. In particular, $\mathbb{D}_\Sigma$ and $\mathbb{D}_T$ have the same domain. We can therefore naturally combine these two relational structures into one relational structure $\mathbb{D}$ over the schema $\Sigma \cup T$. It is obvious by construction that $\mathbb{D} \in \mathcal{C} \odot \mathcal{F}$ and that

$$\Sigma(\mathbb{D}) = \mathbb{D}_\Sigma, \qquad T(\mathbb{D}) = \mathbb{D}_T.$$

It follows from (16) and (17) that

$$\mathbb{A}_1 \subseteq \mathbb{D}, \qquad \mathbb{A}_2 \subseteq \mathbb{D}$$

is a solution to (13).

We now argue that the blowup function for $\mathcal{C} \odot \mathcal{F}$ is the same as the blowup function for $\mathcal{C}$. For any $\mathbb{A} \in \mathcal{C} \odot \mathcal{F}$, if $\mathbb{A}$ is generated by S, then $T(\mathbb{A})$ is generated by S. This shows that for all $n \in \mathbb{N}$,

$$\text{blowup}_{\mathcal{C}}(n) \leq \text{blowup}_{\mathcal{C}\odot\mathcal{F}}(n).$$

Conversely, if $\mathbb{A}_T \in \mathcal{C}$ is a structure generated by S, then there is a structure $\mathbb{A} \in \mathcal{C} \odot \mathcal{F}$ such that $\mathbb{A}_T = T(\mathbb{A})$. Then $\mathbb{A}$ is also generated by S. This shows that for all $n \in \mathbb{N}$,

$$\text{blowup}_{\mathcal{C}}(n) \geq \text{blowup}_{\mathcal{C}\odot\mathcal{F}}(n).$$

□

In view of applying Corollary 6.3 via Lemma 7.4 we stress that the embedding functions of $\mathcal{C} \otimes \mathbb{F}$ and $\mathcal{C} \odot \mathbb{F}$ are polynomial in the one of $\mathcal{C}$ and the one of $\mathbb{F}$ and that their schema is simply the disjoint union of the schema of $\mathcal{C}$ and the one of $\mathbb{F}$.

Hence for $\mathbb{F}$ such as $\langle \mathbb{N}, \sim \rangle$ or $\langle \mathbb{Q}, < \rangle$, both homogeneous with a polynomial time embedding decision, it follows immediately from Proposition 7.7 that we can assume without loss of generality that our databases contain these numerical values and that our systems make use of the corresponding numerical predicates.

$\langle \mathbb{N}, < \rangle$ is not homogeneous. However the proof of proposition 7.7 only require that $\mathcal{F}$, the set of finite substructures of $\mathbb{F}$, is a Fraïssé class. As the set of finite substructures of $\langle \mathbb{N}, < \rangle$ is the same as the one of $\langle \mathbb{Q}, < \rangle$, it is a Fraïssé class and therefore we can also assume without loss of generality that our databases contain numerical values from $\mathbb{N}$ and that our systems make use of the linear order.

7.4. HOMs are Semi-Fraïssé

As a simple application consider an extension of Example 7.3. Fix any schema with relations only, and no functions. Suppose that $\mathbb{G}$ and $\mathbb{H}$ are two databases over the same schema. Suppose that $\mathbb{H}$ is some database. Recall that $\text{HOM}(\mathbb{H})$ denotes the class of all databases over the schema of $\mathbb{H}$ that map homomorphically to $\mathbb{H}$. In other words, $\mathbb{G} \in \text{HOM}(\mathbb{H})$ if and only if there is a homomorphism $f : \mathbb{G} \to \mathbb{H}$.

LEMMA 7.9. *If $\mathbb{H}$ is a finite database, then $\text{HOM}(\mathbb{H})$ is a semi-Fraïssé class.*

PROOF. The schema Γ is the schema Σ extended by a family $\{h\}_{h\in\mathbb{H}}$ of unary predicates, one for each element of the domain of $\mathbb{H}$. We may view the database $\mathbb{H}$ as a database $\tilde{\mathbb{H}}$ over the extended schema Γ, where a node $h \in \mathbb{H}$ gets the color h. It is easy to see that $\text{HOM}(\mathbb{H})$ is the Σ-reduct of $\text{HOM}(\tilde{\mathbb{H}})$. To complete the lemma, we prove that $\text{HOM}(\tilde{\mathbb{H}})$ is Fraïssé.

We only show here amalgamation, the other two properties being trivial. Consider an instance $\mathbb{A}_1, \mathbb{A}_2, \mathbb{C}$ of amalgamation. The desired structure $\mathbb{D}$ is simply constructed from the disjoint union of $\mathbb{A}_1$ and $\mathbb{A}_2$ by identifying the images of $\mathbb{C}$. It remains to show that $\mathbb{D} \in \text{HOM}(\tilde{\mathbb{H}})$. This is witnessed by the mapping sending each node of $\mathbb{D}$ to its color. The reader can verify that this mapping is a homomorphism. □

Note that the schema of $\tilde{\mathbb{H}}$ in the proof of Lemma 7.9 contains no function symbols, hence we have $\text{blowup}_{\text{HOM}(\tilde{\mathbb{H}})}(n) = n$.

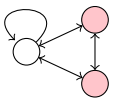
The database $\mathbb{H}$ is called the *template* for the class $\text{HOM}(\mathbb{H})$. Examples of $\text{HOM}(\mathbb{H})$ include n-colorable graphs for every n (when $\mathbb{H}$ is a n-clique).

We show that if a class of databases can be defined as HOM($\mathbb{H}$) for some $\mathbb{H}$, then it admits an algorithm for emptiness of database-driven systems. We actually prove a stronger result where the template is also part of the input.

THEOREM 7.10. *The following problem is* PSPACE*-complete.*

- **Input.** *A template database* $\mathbb{H}$ *and a database-driven register automaton over the schema of* $\mathbb{H}$
- **Output.** *Does the system have an accepting run driven by some database in* HOM($\mathbb{H}$)*?*

Example 7.11. Let $\mathbb{H}$ be the graph below, with nodes colored red or white.



Then a graph $\mathbb{G}$ maps homomorphically to $\mathbb{H}$ if and only if there is no red cycle of odd length in $\mathbb{G}$. On the other hand, the system from Example 3.1 has a $\mathbb{G}$-driven run if and only if there is some red cycle of odd length in $\mathbb{G}$. Therefore, there is no database $\mathbb{G} \in$ HOM($\mathbb{H}$) such that the system has an accepting run driven by $\mathbb{G}$.

The lower bound of Theorem 7.10 follows from Lemma 3.2. Its upper-bound of is an immediate consequence of Corollary 6.3 and the fact that HOM($\mathbb{H}$) is semi-Fraïssé with the same blowup function (Lemma 7.9), same embeddability function, and same arity in the schema. Note that we could also get the same result using database-driven Büchi automata and an EXPSPACE emptiness test for database-driven pushdown automata.

Adding data values. Moreover, using Theorem 7.7, the result of Theorem 7.10 remains valid if each node of the graph carries a unique data value in $\mathbb{N}$ (or any other homogeneous data structure) and the query can test these values using equalities or inequalities (or any other predicate of the homogeneous data structure).

8. XML DOCUMENTS AND REGULAR TREE LANGUAGES

We now turn to the main result of the paper. We consider a class of databases motivated by XML databases. We work with vertex-labeled, unranked and sibling-ordered trees. We use the standard terminology for trees: root, leaf, descendant, ancestor, child, parent, sibling. The *next sibling* of a node x is the first (and therefore unique) sibling after x in document order, which might not exist if x is a rightmost child. The *following sibling* is the transitive (but not reflexive) closure of the next sibling relation. Likewise, each node has at most one *previous sibling*, but possibly many *preceding siblings*. We use the standard notion of regular languages for unranked trees. The automata model is presented in Section 9.1.

It is easy to see that in the presence of a successor relation database-driven systems can simulate counters and are therefore undecidable. See also Section 10.1. For this reason we disallow in our model the use of the child, parent, next sibling and previous sibling relations and only allow relations such as ancestor, descendant, following and preceding sibling. As a matter of fact we can also include the document order[2] and the closest common ancestor function that maps x, y to the node that is a descendant of all common ancestors of both x and y.

We model a tree t as a database, denoted by Treedb(t), whose domain is the nodes of the tree, and which is equipped with the following predicates and functions:

- A unary predicate for every possible node label (there are finitely many labels);

[2]The document order is the order in which the nodes are first reached by a depth-first left-first traversal of the tree

– Binary predicates for document order (denoted by $\leq_{doc}$) and descendant order (denoted by $\preceq_v$);
– A binary function for closest common ancestor, which is denoted by $x \wedge y$. Observe that the descendant relation is defined in terms of this function by a quantifier-free formula:

$$x \preceq_v y \qquad \text{iff} \qquad x = x \wedge y$$

If the set of node labels is A, then the schema above is denoted by TreeSchema(A).

Our main result on trees is that emptiness of database-driven systems is decidable over any regular tree language, even when the description of the regular language is also part of the input.

THEOREM 8.1. *The following problem is decidable:*

– **Input.** *A tree automaton defining a language L of trees labeled by an alphabet A, and a database-driven register automaton $\mathcal{S}$ over* TreeSchema(A).
– **Output.** *Is there a tree $t \in L$ and an accepting run of $\mathcal{S}$ driven by* Treedb(t)*?*

For a fixed tree automaton, the problem is in PSPACE *and there are automaton for which the problem is complete for* PSPACE. *When both the tree automaton and the database-driven register automaton are given on input, the problem is in* EXPSPACE.

The database-driven register automata are only allowed to access the trees through quantifier-free formulas that use the predicates included in TreeSchema(A). Recall that we could also allow them to use existential formulas defined in terms of the predicates in TreeSchema(A). Some navigation predicates for trees, such as child, next sibling, or even simply sibling are not definable this way. We will later show (Section 10) that adding any one of the above three predicates leads to undecidability.

If L is a tree language then we denote by Treedb(L) the set of databases modeling trees in L. The proof of Theorem 8.1 is a consequence of the fact that Treedb(L) is semi-Fraïssé as we will prove in Section 9.

Adding data values. By Proposition 7.7 the result of Theorem 8.1 remains valid if each node of the tree also carries a data value in $\mathbb{F} = \langle \mathbb{N}, \sim \rangle$ (or any other homogeneous data structure) and the query can test these values using equalities or inequalities (or any predicate of the data structure). In the case where the data structure is $\langle \mathbb{N}, \sim \rangle$ the decidability result concerns databases that are also known as *data trees*. The complexity bound is not affected by this extension:

THEOREM 8.2. *The following problem is decidable:*

– **Input.** *A tree automaton defining a language L of trees labeled by an alphabet A, and a database-driven register automaton over the schema* TreeSchema(A) $\cup \{\sim\}$.
– **Output.** *Is there a tree $t \in L$, a labeling λ of t by elements of $\mathbb{N}$, and an accepting run of the system driven by* Treedb(t) $\otimes \lambda$*?*

For a fixed tree automaton, the problem is in PSPACE *and there are automaton for which the problem is complete for* PSPACE. *When both the tree automaton and the database-driven register automaton are given input, the problem is in* EXPSPACE.

Remark 8.3. Theorem 8.2 builds on Theorem 8.1 and Proposition 7.7 with $\mathbb{F} = \langle \mathbb{N}, \sim \rangle$. But any homogeneous $\mathbb{F}$ with PSPACE embeddability would also do the job. This includes $\langle \mathbb{Q}, < \rangle$ and even $\langle \mathbb{N}, < \rangle$ as argued before.

Remark 8.4. By repeatedly applying Proposition 7.7, we could have a stronger version of Theorem 8.2, where each tree node stores k data values that can be compared for equality.

In terms of complexity the results are polynomial in k which appears within the size of the schema.

9. REGULAR TREE LANGUAGES

In this section, we prove Theorem 8.1, which is the main technical result of the paper. The theorem says that emptiness is decidable for database-driven systems over regular tree languages. We fix a regular language L of trees labeled by an alphabet A.

We start by defining a class $\mathcal{C}$ that represents substructures of runs, with good properties.

PROPOSITION 9.1. *For all regular tree language L over the alphabet A, there exists a class of finite structures $\mathcal{C}$ over a schema* $\mathrm{RunSchema}(A) \supseteq \mathrm{TreeSchema}(A)$ *such that the following properties hold.*

(1) *$\mathcal{C}$ is closed under substructures and under amalgamation,*
(2) $\mathrm{substructures}_{\mathrm{fin}}(\mathrm{Treedb}(L)) = \mathrm{substructures}_{\mathrm{fin}}(\mathrm{TreeSchema}(A)(\mathcal{C}))$,
(3) *the blowup function of $\mathcal{C}$ is $n \mapsto c \cdot n$, where c is exponential in the size of the automaton recognizing L,*
(4) *$\mathcal{C}$ has membership decidable in* PSPACE,
(5) *The schema* $\mathrm{RunSchema}(A)$ *consists of function and relation symbols of arity at most two, and the number of those symbols is exponential in the size of A.*

The above proposition implies Theorem 8.1, using Lemma 7.4 and Corollary 7.2. In the rest Section 9, we prove Proposition 9.1.

9.1. Tree automata and their components

We begin by presenting the model of tree automata that we use. This model is equivalent to the models used in the literature [Comon et al. 2002].

A *tree automaton* consists of:

- An input alphabet A. The automaton is used to accept or reject trees labeled by A.
- A set of states Q.
- The automaton has a distinguished *root state*, which is the only state allowed at the root.
- With each state $q \in Q$ there is an associated regular language $L_q \subseteq Q^*$, represented by a nondeterministic finite automaton $\mathcal{A}_q$ over the alphabet Q. Without loss of generality we assume the automaton $\mathcal{A}_q$ to have the property that for every of its state h there is a unique letter $q \in Q$ which can be read in state h.

A run over a tree labeled by elements of A is a labeling of the nodes and edges of the input tree, which satisfies certain local consistency requirements described below. Every node is labeled by exactly one state $q \in Q$. We adopt the convention that for each state q there is precisely one label $a \in A$ which can be read in state q, i.e. if a node is labeled by the state q, then it is also labeled by the letter a. Furthermore, if a node x has label q, then every edge leaving from x has some label h which is a state of the automaton $\mathcal{A}_q$, and the sequence of labels on the edges leaving from x forms an accepting run of $\mathcal{A}_q$ over the sequence formed by the labels of the children of x.

A tree is accepted by the automaton if it admits some run whose root is labeled by the root state. Let us fix for the rest of Section 9 a tree automaton as described above, which recognizes a tree language L.

Descendant components. Define a binary relation $\to_D$ on the states of the automaton which contains all pairs p, q such that in some run, the state p appears in some node which is an ancestor of a node with state q. This relation can be computed in polynomial time from a given automaton. We may therefore enforce (in polynomial time) that the automaton contains only useful states, i.e. if q is a state of the automaton and r is the root state, then $r \to_D q$.

We use the name *descendant component* for a strongly connected component Δ of the relation $\to_D$. We adopt the convention that when a state is not reachable from itself by $\to_D$, then it still forms a (singleton) descendant component.

We distinguish two kinds of descendant components:

- A descendant component Δ is called *branching* if in some run, some node with state in Δ has two children with states in Δ.
- A descendant component Δ is called *linear* otherwise. This means that in every run, every node with state in Δ has at most one child with a state in Δ.

Horizontal components. Fix a state q and its associated automaton $\mathcal{A}_q$. Define a binary relation $\to_H^q$ on the states of the automaton $\mathcal{A}_q$ which contains all pairs h, h' such that in some (accepting) run of $\mathcal{A}_q$, the state h appears before the state h'. This relation can be computed in polynomial time from a given automaton. We use the name *horizontal component* for a strongly connected component Γ of the relation $\to_H^q$ (where q is implicit). Again, we adopt the convention that when a state is not reachable from itself by $\to_H^q$, then it still forms a (singleton) descendant component. We call a horizontal component *repeating* if it is not of this form, i.e. if any two states h, h' in this component can be reached from each other by some nonempty word.

The set of horizontal components of the automaton $\mathcal{A}_q$ inherits from $\mathcal{A}_q$ the structure of a directed acyclic graph: there is an edge between two horizontal components if there is a transition from a state in the first component to a state in the second component. A *history* is a simple directed path Π in the directed acyclic graph of horizontal components of $\mathcal{A}_q$, which starts in a component containing an initial state and ends in a component containing an accepting state of the automaton $\mathcal{A}_q$.

Fix a run π and a node x in π with state q as label. Let $h_1, \ldots, h_n$ be the sequence of labels at the edges leaving from x in π. Then the sequence $h_1, \ldots, h_n$ induces a history Π in the directed acyclic graph of horizontal components of the automaton $\mathcal{A}_q$, by simply listing (without repetitions) the horizontal components visited by $h_1, \ldots, h_n$. We call Π the *history* of x.

9.2. The class $\mathcal{C}$

We are now ready to define the class $\mathcal{C}$, which contains databases representing parts of runs.

The pointers. For a run π, we define $\mathrm{Rundb}(\pi)$ to be the following database.

- It contains the relations from $\mathrm{Treedb}(t)$ for the tree t underlying π: the node labels, the descendant order, the document order, and the closest common ancestor function.
- There is a constant representing the root of π.
- For each state q and history Π in $\mathcal{A}_q$ there is a unary relation $\mathit{history}_\Pi(x)$ marking a node x if it has state q and history Π in π.
- For each state q and horizontal component Γ of $\mathcal{A}_q$ there is one binary relation $\mathit{direction}_\Gamma(x, y)$ which relates a node x with a node y if y is a descendant of x, the state in x is q and the first edge on the path from x to y has a label in the component Γ.
- For each descendant component Δ, there is a unary function $\mathit{closest}_\Delta(x)$, whose value is the first node different from x on the path from x to the root that has state in Δ. If there is no such node, then the function is "undefined", which is encoded by $\mathit{closest}_\Delta(x) = x$.
- For each state q and non-repeating horizontal component Γ of $\mathcal{A}_q$ there is one unary function $\mathit{unique}_\Gamma(x)$ which to a node x with state q assigns the child which is the target of the unique edge leaving from x whose label is in the component Γ, under the condition that the state in this child is in a different descendant component than q. If x does not have state q, or does not have a unique edge with label in Γ, or the child at the end of that edge has state

in the same descendant component as q, then $unique_\Gamma(x)$ is "undefined", which is encoded by $unique_\Gamma(x) = x$.

– Suppose that x is a node whose state is in a linear descendant component Δ. Then $descendantmost(x)$ maps x to the unique descendant of x that has a state in Δ but none of its children has a state in Δ. If the state of x is not in a linear descendant component, then the function is "undefined", which is encoded by $descendantmost(x) = x$.

This ends the definition of the database $\mathrm{Rundb}(\pi)$. We define $\mathrm{RunSchema}(A)$ as the schema of such structures. Clearly it consist of symbols of arity at most two, whose number is exponential in the size of A (due to the predicates $history_\Pi(x)$).

The class $\mathcal{C}$. Define $\mathcal{C}$ to be the closure under substructures of the class

$$\{\mathrm{Rundb}(\pi) : \pi \text{ is a run of the automaton}\}.$$

In other words, a database belongs to $\mathcal{C}$ if it can be extracted from a run so that nodes are extracted together with the values of their pointers. Clearly, since every tree in L is the projection over $\mathrm{TreeSchema}(A)$ of a structure $\mathrm{Rundb}(\pi)$ for a run π, it follows that the second condition of Proposition 9.1 holds. Also, $\mathcal{C}$ is closed under substructures by definition.

We will call pre-runs the structures of $\mathcal{C}$.

To prove Proposition 9.1 it remains to prove the following lemmas.

LEMMA 9.2. *The blowup function for $\mathcal{C}$ is $n \mapsto c \cdot n$, where the constant c is exponential in the state space Q.*

LEMMA 9.3. *Membership in $\mathcal{C}$ can be decided in* PTIME.

LEMMA 9.4. *$\mathcal{C}$ is closed under amalgamation.*

We proceed to the proof of the first lemma.

PROOF OF LEMMA 9.2. Let F be the function symbols available in the schema, i.e. F is the closest common ancestor function *cca*, as well as the pointer functions

$$\underbrace{unique_\Gamma, descendantmost}_{P_\downarrow}, \underbrace{closest_\Delta}_{P_\uparrow}$$

for the possible choices of Γ and Δ. Let us write $P \subseteq F$ for the pointer functions, and partition P into two parts: the part $P_\uparrow$ which contains the $closest_\Delta$ pointer functions, and the part $P_\downarrow$ which contains the remaining pointer functions.

Fix a structure ρ of $\mathcal{C}$. For a set X of nodes in ρ and a subset $G \subseteq F$, we write $[X]^G$ for the smallest set of nodes that contains X and is closed under applying functions from G. Our goal is to prove that for the set F of all functions, the size of $[X]^F$ is at most c times the size of X, where the multiplicative constant c is at most exponential in the state space of the automaton.

We use the two following claims which show that there is a natural order in applying the functions. The proofs are a straightforward case analysis based on simple observations such as within a any structure of $\mathcal{C}$ we have: $closest_{\Delta'}(descendantmost(closest_\Delta(x))) = closest_{\Delta'}(x)$.

CLAIM 1. $[X]^F = [[X]^{cca}]^P$

CLAIM 2. $[X]^P = [[X]^{P_\uparrow}]^{P_\downarrow}$.

Combining the two claims, we see that $[X]^F$ is obtained by first closing X under *cca*, then closing the resulting set under $P_\uparrow$, and finally closing the resulting set under $P_\downarrow$. Closing a set under *cca* makes it grow by at most a multiplicative factor of two. Closing a set under $P_\uparrow$

makes it grow by at most a multiplicative factor of the number of descendant components. Finally, closing a set under $P_\downarrow$ makes it grow by at most a multiplicative factor that is exponential in the state space of the automaton. The second and third statements are because applying a function from $P_\uparrow$ or $P_\downarrow$ that is not a self-loop requires a change of component. □

We now turn to the complexity of the membership problem. Note that it would be sufficient to prove PSpace. But as it is merely tree automata evaluation, we get PTime.

Proof of Lemma 9.3. Let $\mathbb{A}$ be a database over the schema RunSchema(A). The following are necessary and sufficient conditions for the existence of a run π such that $\mathbb{A}$ is a substructure of Rundb(π).

- The descendant order, the document order, the closest common ancestor function, consistently describe a tree whose root is the root constant.
- Each note has a label in Q and the root has the root state as label.
- Each node x is labeled by predicate $history_\Pi$, for some history Π of $\mathcal{A}_q$ where q is a label of x.
- If x has history Π, each descendant y of x must satisfy exactly one relation $direction_\Gamma(x, y)$ for some Γ occurring in Π. Moreover, for Γ occurring in Π, the set of y such that $direction_\Gamma(x, y)$ holds is either empty or forms a sibling sequence of subtrees of x. The sequences are arranged in the order given by Π.
- The states q labeling the nodes of the tree correspond to a valid run of the automaton consistent with the histories. This can be done in non deterministic LogSpace by guessing for each node x of label q and history Π the sequence of states in $\mathcal{A}_q$ while testing it is consistent with Π.

The above conditions can be verified in time polynomial in the size of the domain of $\mathbb{A}$. □

It remains to prove Lemma 9.4, i.e. that $\mathcal{C}$ is closed under amalgamation. We need a notion for describing operations on trees, which amounts to replacing one part of a tree by another tree.

Contexts and replacements. A run or a pre-run σ together with a distinguished set $y_1, \ldots, y_n$ of nodes which are mutually incomparable with respect to the descendant relation is called a *multi-context* of arity n. The nodes $y_1, \ldots, y_n$ are called the *ports* of the multi-context σ; the ports are naturally ordered by the document order. Multi-contexts of arity 0 are simply pre-runs, multi-contexts of arity 1 are *contexts*, and multi-contexts of arity 2 are *bi-contexts*. Multi-contexts can be composed in the natural way, by replacing the subtree at a specified port by another multi-context.

Given a run or a pre-run σ and nodes $x, y_1, \ldots, y_n$ in σ such that x is an ancestor of $y_1, \ldots, y_n$ and $y_1, \ldots, y_n$ are mutually incomparable, we define the multi-context $\sigma[x; y_1, \ldots, y_n]$ as the substructure σ rooted at x with the nodes $y_1, \ldots, y_n$ as the ports. In the extreme case where $n = 0$, then $\sigma[x;]$ denotes the substructure of σ induced by the subtree rooted at x.

Given a run or a pre-run σ and a node x in σ, given a context δ define "σ with δ *inserted* above x" the structure composing the following three contexts $\sigma[r; x]$, δ and $\sigma[x;]$.

We will also use the following terminology. Given a pre-run ρ we say that an edge from x to y has label Γ in ρ if $direction_\Gamma(x, y)$ holds in ρ.

Proof of Lemma 9.4. By Lemma 4.9, it suffices to show that $\mathcal{C}$ is closed under inclusion amalgamation. Consider an instance of inclusion amalgamation in the class $\mathcal{C}$, i.e. two consistent pre-runs ρ_1 and ρ_2 of $\mathcal{C}$. Let ρ_0 be common part, i.e. the intersection of the two databases, which is a substructure of both, so it also belongs to $\mathcal{C}$.

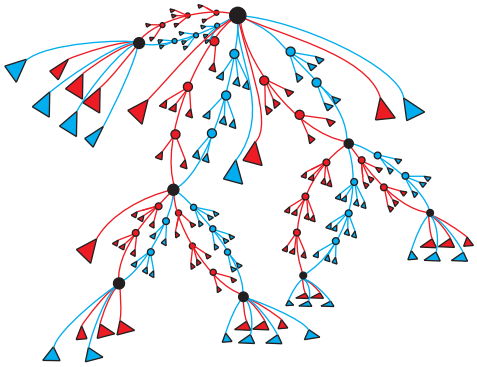


Fig. 1. The pre-runs ρ_1 and ρ_2. The red parts are fragments of the pre-run ρ_1, and the blue parts are fragments of the pre-run ρ_2. The black nodes are the nodes of ρ_0. Triangles represent subtrees with no node in ρ_0.

An impressionistic illustration of the situation is depicted in Figure 1.

We need to show a database in $\mathcal{C}$ that contains both ρ_1 and ρ_2 as substructures. The proof is by induction on the number of nodes in ρ_1 which are not in ρ_0. In the induction base, ρ_1 is a substructure of ρ_2, and we already have a solution to amalgamation.

Now consider the inductive step.

In the inductive step, there is a node y which is in ρ_1 but not in ρ_0. Choose the node y so that all its ancestors are already in ρ_0. Let x be the parent of y. There are two cases, depending on whether or not y has a descendant in ρ_1 which belongs to ρ_0. In each case we modify the pre-run ρ_2 to a pre-run ρ_2', reducing the inductive parameter.

Case 1: *The node y has a descendant which belongs to ρ_0.*

Let x' be the unique child of x in ρ_0 which is a descendant of y in ρ_1. The fragments of the pre-runs ρ_1 and ρ_2 between x and x' are depicted in Figure 2, on the left.

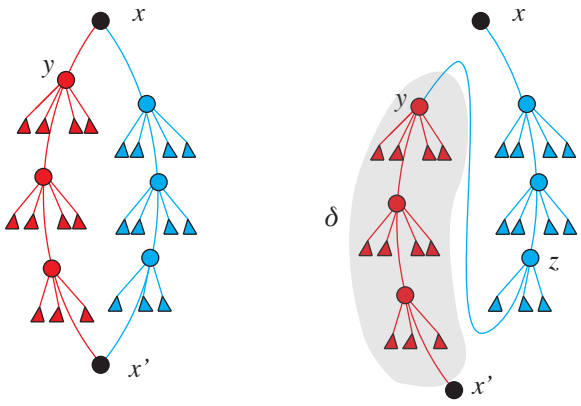


Fig. 2. Construction of ρ_2' in Case 1. *Left:* The fragments of the pre-runs ρ_1 and ρ_2 between the nodes x and x'. The nodes of the pre-run ρ_1 are marked red, and the nodes of the pre-run ρ_2 are marked red. *Right:* The pre-run ρ_2', obtained from run ρ_2 by inserting the context δ above x'.

Let δ be the context $\rho_1[y; x']$. Let $\hat{\rho}_2$ be constructed by inserting δ above x' as depicted in Figure 2, on the right. We now set the remaining relations and functions in order to view $\hat{\rho}_2$ as the desired pre-run ρ_2'.

The key observation is the following fact.

FACT 9.5. *Both in ρ_1 and in ρ_2, all nodes between x and x' have states in the same descendant component as the state in x'.*

PROOF. We show the argument for nodes v in ρ_1; the argument for nodes in ρ_2 is the same. Let v be a node different from x which lies on the path from x to x' in ρ_1. Let Δ be the descendant component of the state at the node v in ρ_1. Suppose that the state at x' is not in Δ. Then in ρ_1, $closest_\Delta(x')$ points to a node z which is strictly between x and x'. Since ρ_0 is a substructure of ρ_1, it follows that $closest_\Delta(x')$ is also equal to z in ρ_0. But in ρ_0 there is no node which is strictly between x and x' – a contradiction. □

By our assumption ρ_2 and ρ_1 are in $\mathcal{C}$. Therefore they are substructures of $\mathrm{Rundb}(\pi_1)$ and $\mathrm{Rundb}(\pi_2)$ for some runs π_1 and π_2. It follows from Fact 9.5 above that y and x' are in the same descendant component. Hence there is a run π which contains two nodes z and z', with z' being the descendant of z, and such that the state at z in π is the same as the state of x', and the state at z' in π is the same as the state of y. Let σ be the context $\pi[z; z']$. Let δ' be the context $\pi_1(y; x')$. Let σ' be the context composing σ and δ'. We extend π_2 by inserting σ' above x' as depicted in Figure 3 below. It is easy to verify that this yields a valid run π_2' that contains all nodes of $\hat{\rho}_2$. Hence the substructure ρ_2' induced by those nodes is our desired pre-run.

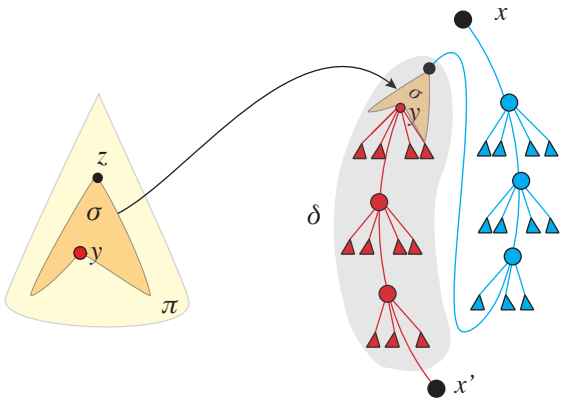


Fig. 3. *Left:* The run π and its fragment σ. *Right:* The run π_2 extended to a run by inserting the fragment σ above y.

It is now easy to verify that ρ_2' extends ρ_2, that ρ_2' is still consistent with ρ_1 and that ρ_1 and ρ_2' share at least one extra node, namely y (in fact all nodes in δ).

Case 2: *The node y has no descendants which belong to ρ_0.*

Let $\Pi = \Gamma_1 \dots, \Gamma_n$ be such that $history_\Pi(x)$ holds in ρ_1. Therefore $history_\Pi(x)$ also holds in ρ_2, as x belongs to both structures. Hence labels of the edges leaving x form a subsequence of Π.

There is some i such that the label of the edge from x to y in ρ_1 belongs to Γ_i. Define ρ_2' as the result of inserting $\rho_1[y;]$ to ρ_2 as a child of x, so that it follows all children whose edge labels are in Γ_j with $j < i$ and precedes all children of x whose edge labels are in Γ_j with $j \geq i$ (see Figure 4). Moreover for all nodes z in $\rho_1[y;]$ we have $direction_{\Gamma_i}(x, z)$ in ρ_2, the remaining functions being defined in the natural way.

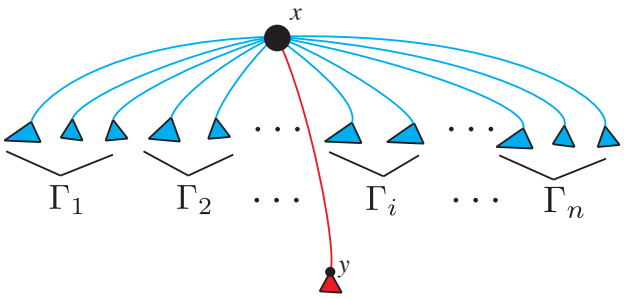


Fig. 4. The fragment of the run ρ_2 below the node x, with the subtree of y added in the appropriate place.

We need to show that ρ_2' is a pre-run. We distinguish between two cases, according to the following fact.

FACT 9.6. *At least one of the following cases occurs:*

- *The label of the edge from x to y in ρ_1 is a repeating horizontal component Γ.*
- *The states in the nodes x and y in ρ_1 are in the same descendant component Δ, which is branching.*

PROOF. Suppose first that the states in x and y in ρ_1 belong to the same descendant component Δ. We show that Δ is branching. Indeed, otherwise Δ would be a linear component, and $\mathit{descendantmost}(x)$ would point to some descendant z of y in ρ_1, and therefore also in ρ_0, and so z would belong to ρ_0, contradicting the assumption that y has no descendants in ρ_0.

Now suppose that the states in x and y belong to different descendant components. Let Γ be the label of the edge from x to y in ρ_1. If Γ was a non-repeating horizontal component, then $\mathit{unique}_\Gamma(x)$ would be equal to y, and so y would belong to ρ_0, a contradiction. □

CLAIM 3. *ρ_2' is a pre-run.*

PROOF. By hypothesis ρ_1 and ρ_2 are substructures of $\mathrm{Rundb}(\pi_1)$ and $\mathrm{Rundb}(\pi_2)$ for some runs π_1 and π_2.

We show how to extend ρ_2' to a run. Consider the two cases described in Fact 9.6.

- *The label Γ of the edge from x to y in ρ_1 is a repeating horizontal component.* Note that by construction the label h of the edge from x to y in π_1 is in Γ. Let $h_1, \ldots, h_n$ denote the labels of the edges leaving from x in π_2 and let $\Gamma_1, \ldots, \Gamma_n$ be their corresponding horizontal component. Let t be $\pi_1[y;]$, i.e. the subrun of π_1 that reaches y. Recall that ρ_2' has been constructed from ρ_2 by attaching $\rho_1[y;]$ to x right before those edges of label Γ. Let i be minimal such that h_i belongs to Γ. In particular $\Gamma = \Gamma_i$. By construction h is reachable from h_{i-1} in $\mathcal{A}_q$ because it belongs to Γ_i. Moreover h_i is reachable from h because both belongs to the same strongly connected component Γ_i of $\mathcal{A}_q$. Hence the sequence $h_1, \ldots, h_n$ can be extended into a sequence forming an accepting run of $\mathcal{A}_q$ and having $h_1, \ldots, h_{i-1}, h, h_i, \ldots, h_n$ as subsequence. Using this fact, by adding to π_2 the attaching the tree t to x at the corresponding place in this sequence and by adding appropriate subruns for the other extra nodes of the sequence (here we use the assumption that the tree automaton contains only useful states) we obtain a run π_2' such that ρ_2' is a substructure of $\mathrm{Rundb}(\pi_2)$.
- *The states in the nodes x and y in ρ_1 are in the same descendant component Δ, which is branching.* We furthermore assume that the previous case does not hold, i.e., the label Γ of the edge from x to y in ρ_1 is a non-repeating horizontal component. Let h be such that $\Gamma = \{h\}$. In particular, since π_2 is a run and the history of the node x in π_1 and

π_2 are the same, it follows that there is a unique edge of label h leaving from x in π_1 and in π_2. Let z_1 and z_2 be the corresponding endpoints of those edges. Note that by construction y is a descendant of z_1. Since in a run the edge label determines the state at the endpoint, it follows that the states in z_1 and in z_2 are the same, say, q. Moreover, q is in the same branching descendant component Δ as the state in x. It follows that there is a run π which contains the node u with the same state and history as x in ρ_0, and two incomparable descendants v and v', both with state q. Let σ denote the bi-context $\pi[u; v, v']$. Let σ_1 be $\pi_1[z_1;]$, the subtree of π_1 rooted at z_1. Let σ' denote the context constructed by plugging σ_1 in the first port of σ. Let π_2' be constructed from π_2 by replacing $\pi_2[x; z_2]$ by σ'. It is easy to verify that π_2' is a run and that ρ_2' is a substructure of $\mathrm{Rundb}(\pi_2')$.

□

It is easy to see that ρ_2' extends ρ_2. Moreover ρ_2' and ρ_1 share at least one extra node, namely y (actually all the subtree of y).

Hence it remains to show:

CLAIM 4. *ρ_2' is consistent with ρ_1.*

PROOF. We only show that the *unique* functions are consistent in ρ_1 and in ρ_2'; the cases of the remaining relations and functions are evident.

Let Γ be such that $\mathit{direction}_\Gamma(x, y)$ holds in ρ_1. The value $\mathit{unique}_\Gamma(x)$ in ρ_1 must be equal to x – otherwise it would be equal to y and so y would belong to ρ_0, a contradiction. It follows that either Γ is a repeating horizontal component, or the descendant component of the state in x and of the state in y are equal.

In either case, $\mathit{unique}_\Gamma(x)$ is equal to x in the ρ_1, ρ_2 and ρ_2'. □

The above analysis ends the inductive proof of Lemma 9.4, ending the proof of Proposition 9.1 and of Theorem 8.1.

10. UNDECIDABLE MODELS

We consider in this section several ways of extending the model. In most cases, the extensions lead to undecidability. For instance, if the trees are additionally equipped with the child relation or the sibling relation, then the emptiness problem becomes undecidable, even for a fixed tree language. A more interesting question is what happens if we extend the expressive power of the logics used for describing the transitions of the system. These extensions also quickly lead to undecidability.

10.1. Child and sibling axes

Adding axes such as *next sibling* or *child* leads to undecidability. The reason is that already for unary words with the successor relation on positions, we get undecidability. More precisely, a unary word w can be viewed as a structure whose domain is the set $1, 2, \ldots, |w|$ of positions of w, and $\mathit{succ}(x, y)$ holds if $y - x = 1$.

FACT 10.1. *Let L be any infinite set of words over the unary alphabet, viewed as structures over the schema consisting of the binary symbol succ. Then, the following problem is undecidable.*

- **Input.** *A database-driven register automata over the schema consisting of succ.*
- **Output.** *Is there a word $w \in L$ and an accepting run of the system driven by the word w?*

PROOF SKETCH. Using one register, the system can simulate a counter of a counter machine: a transition can increment or decrement the counter using the relation *succ*. There

are no zero tests, but this can be simulated by keeping one register z that is never changed (using $z_{\text{old}} = z_{\text{new}}$ as a conjunct in all rules); then a zero test of the counter is simulated by the formula $x = z$. Since the halting problem is undecidable for two-counter machines, the fact follows. □

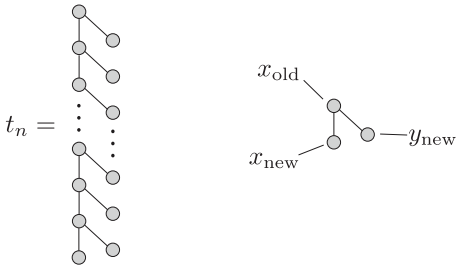
It follows immediately from Fact 10.1 that in the presence of the *child* or *next sibling* axis it is undecidable whether a database-driven has an accepting run.

The sibling axis. We show that even extending the set of predicates by the sibling relation also leads to undecidability. Formally, we model a tree t as a database with two predicates: the closest common ancestor $\wedge$ and the transitive, symmetric and irreflexive binary *sibling* relation (the document order nor the unary predicates are needed for this undecidability result).

FACT 10.2. *There exists a regular tree language L over a unary alphabet, such that the following problem is undecidable:*

- **Input.** *A database-driven register automata over the schema consisting of $\wedge$ and sibling.*
- **Output.** *Is there a tree $t \in L$ and an accepting run of the system driven by the tree t?*

PROOF SKETCH. The language L is defined as the set of trees of the form t_n, where $n \in \mathbb{N}$ and t_n is the tree depicted in the left-hand side of the figure below, of height n.



The reduction is again from counter machines. We show that a system can simulate a counter using a register x.

To simulate incrementation of the counter, the machine uses an auxiliary register y (see right-hand side of the figure above), and follows a transition whose guard is the following formula:

$$\big(x_{\text{old}} = (x_{\text{new}} \wedge y_{\text{new}})\big) \quad \wedge \quad \mathit{sibling}(x_{\text{new}}, y_{\text{new}})$$

These conditions guarantee that x_{new} is a child of x_{old} (they do not guarantee that x_{new} is the left child, as is the case in the figure, but this is not necessary).

Decrementation of a counter is obtained by swapping "old" with "new" in the guard. As in the proof of Fact 10.1, using additional counters, one can simulate zero tests. This way, a database-driven system using the predicates $\wedge$ and the successor relation can simulate a counter machine. □

Remark 10.3. We don't know whether emptiness is decidable for database-driven systems over the schema consisting of the sibling relation, the document order and the vertical order, but not the closest common ancestor.

10.2. Rules that are not existential

A legitimate question is whether one can extend the expressivity of our model by extending the power of the formulas guarding the transitions of the system, for instance by allowing first-order formulas, while preserving the decidability of the emptiness problem.

We have seen in Section 3.5 that systems with transitions guarded by existential formulas can be simulated by systems where the transitions are guarded by quantifier-free formulas. However, already allowing boolean combinations of existential formulas quickly leads to undecidability. For instance, in the tree case, this is a consequence of Fact 10.1. Indeed, using boolean combinations of existential formulas, one can define the *child* axis:

$$\mathit{child}(x,y) \quad \Longleftrightarrow \quad x \preceq_v y \quad \wedge \quad \neg\exists z: \quad x \prec_v z \prec_v y$$

10.3. Data tree patterns

We also considered the setting where the queries are data tree patterns. A data tree pattern selects data values within a tree, depending on the existence of nodes whose positions verify the tree pattern (we use an injective semantics for tree patterns, where each node of the tree pattern must match a different node of the tree). With our terminology, a tree pattern is a special case of an existential formula; the hope being that systems using boolean combination of tree patterns would be decidable.

To make this setting fit into our formalism, we assume the system is over $\mathrm{TreeSchema}(A) \cup \{\sim\}$ and has rules guarded by boolean combinations of formulas of the following form (called tree pattern formulas):

$$\delta(\bar{x}_{\mathrm{new}}, \bar{x}_{\mathrm{old}}) \quad : \quad \exists^{\neq} v_1, v_2, \ldots, v_l \quad \varphi(v_1, \ldots, v_l),$$

where the notation $\exists^{\neq}$ implies that the nodes $v_1, \ldots, v_l$ are pairwise distinct (to reflect the injective semantics of tree patterns), and φ is a conjunction of conjuncts of the following three possible forms:

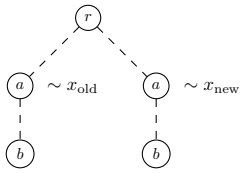
$$v_i \sim x_j, \qquad v_i \preceq_v v_j, \qquad \lambda(v_i).$$

Note that the restriction on φ implies that a formula δ cannot (directly) tell whether two registers x and y of the system point to the same node; it can only test whether they have the same datavalue.

Example 10.4. Consider trees labeled by $\{a, b, r\}$. The following tree pattern:

$$\exists^{\neq} v, l_a, l_b, r_a, r_b. \ \Big(a(l_a) \wedge b(l_b) \wedge a(r_a) \wedge b(r_b) \wedge r(v)\Big) \wedge \\ (v \preceq_v l_a \preceq_v l_b) \ \wedge \ (v \preceq_v r_a \preceq_v r_b) \ \wedge \ (l_a \sim x_{\mathrm{old}}) \quad \wedge \quad (r_a \sim x_{\mathrm{new}})$$

can be graphically represented as follows (dashed lines represent descendant relationships):

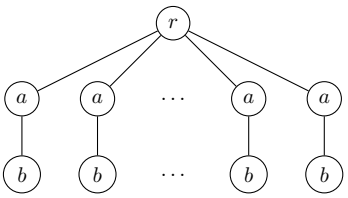


THEOREM 10.5. *There is a language of A-labeled trees L such that the following decision problem is undecidable.*

- **Input.** *A database-driven register automata over the schema $\{\preceq_v, \sim, \{a\}_{a\in A}\}$ where transitions are boolean combinations of tree pattern queries.*
- **Output.** *Is there a data tree $t \in L \otimes \langle \mathbb{N}, \sim \rangle$ and an accepting run of the system driven by the tree t?*

PROOF.

The language L consists of trees of the following form. I.e. unranked trees of depth three where only the root may have more than one child.



The general idea is to encode runs of counter (or Minsky) machines. More precisely, for a given counter machine M with counters C we will construct a database-driven register automata S_M over the schema $\{\preceq_v, \sim, \{a\}_{a\in A}\}$ such that

> M has an accepting run if and only if S_M has an accepting run driven by some tree $t \in L \otimes \langle \mathbb{N}, \sim \rangle$.

Since the halting problem for counter machines is undecidable (even with two counters), this will imply that emptiness of our systems is also undecidable.

We will describe a mechanism which allows to simulate each counter from C separately in S_M – this simulation allows increasing and decreasing counters, and testing whether two counters are equal to each other.

The states Q of S_M are precisely the states of the machine M. For simplicity, we assume that the transition relation of the machine M is such that at each step, only one counter is accessed (i.e. its value is incremented, decremented, or checked for equality with 0).

Consider a tree $t \in L \otimes \langle \mathbb{N}, \sim \rangle$. Let $t_1, t_2, \ldots, t_n$ be all the maximal subtrees of t not containing the root of t: these trees have a root labeled a and a unique child of label b, both nodes carrying a data value in $\mathbb{N}$.

We will say that t_j is a *successor* of t_i (equivalently, that t_i is a *predecessor* of t_j), if the a-node u of t_j and the b-node v of t_i have the same datavalues, i.e.

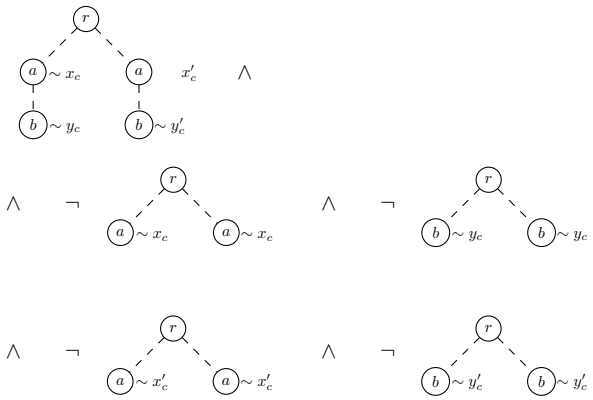
$$u \sim v.$$

Note that a priori, t_i may have many successors and many predecessors.

For each counter $c \in C$, the system S_M will store in its variables, at any moment, two nodes x_c, y_c. The invariant maintained during the run is:

- There is precisely one subtree t_i of t whose a-node has the same value as x_c;
- This unique subtree t_i has b-node whose value is the same value as y_c;
- There is precisely one subtree t_j of t whose a-node has the same value as in y_c.

Because of the uniqueness described in the first item above, we can say that the nodes (x_c, y_c) *define* the subtree t_i. We will now describe how the system S_M can enforce, by performing a suitable transition inc_c, that if the values (x_c, y_c) before the transition inc_c define the subtree t_i, then after performing the transition inc_c, the new values (x'_c, y'_c) will define a subtree t_j such that t_j is the unique successor of t_i and t_i is the unique predecessor of t_j. In particular $x'_c = y_c$. This is done by verifying the matching in t of the following boolean combination of conjunctive queries (with the natural semantics, as illustrated in Example 10.4):



Satisfaction of the above formula, together with $x'_c = y_c$ guarantees that (x'_c, y'_c) encodes the successor of the tree encoded by (x_c, y_c), and that the invariant is maintained. This way, we simulate an increment of the counter c. In a very similar way, we can simulate decrementing counter c. Zero tests can be simulated by comparing to an extra counter which is never incremented nor decremented.

It is easy to verify that the system S_M has an accepting run (driven by some database $t \in L$) if and only if M has an accepting run. Therefore, by using boolean combinations of conjunctive queries and six variables – one pair for each counter, where a third counter is used to simulate zero tests – our systems can simulate two-counter machines. □

11. CONCLUSIONS

We have exhibited a generic framework for studying automata interacting with a database. We have shown that decidability results could be derived easily assuming Fraïssé or semi-Fraïssé with standard complexities for static analysis. Most classes studied in the literature falls in our setting and we have added one: XML documents with data values.

We believe that the complexities of our results are tight. This isleft for future work.

REFERENCES

Mikołaj Bojańczyk, Luc Segoufin, and Szymon Toruńczyk. 2013. Verification of database-driven systems via amalgamation. In *Proc. Symp. on Principles of Database Systems (PODS)*, Richard Hull and Wenfei Fan (Eds.).

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2002. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata. (2002). release October, 1rst 2002.

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

Alin Deutsch, Liying Sui, and Victor Vianu. 2007. Specification and verification of data-driven Web applications. *J. Comput. Syst. Sci.* 73, 3 (2007), 442–474. `https://doi.org/10.1016/j.jcss.2006.10.006`

Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. 2006. A system for specification and verification of interactive, data-driven web applications. In *Intl. Conf. on Management of Data (SIGMOD)*. `https://doi.org/10.1145/1142473.1142584`

Wilfrid Hodges. 1997. *A shorter model theory*. Cambridge Univerity Press.

Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. 2001. Semantic Web Services. *IEEE Intelligent Systems* 16, 2 (2001), 46–53. `https://doi.org/10.1109/5254.920599`

Luc Segoufin and Szymon Toruńczyk. 2011. Automata based verification over linearly ordered data domains. In *Intl. Symp. on Theoretical Aspects of Computer Science (STACS)*.

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