# Verification of database-driven systems via amalgamation

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

## ABSTRACT

We describe a general 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 either the database or 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

## Keywords

Database-driven Systems, Register Automata, Amalgamation, Fraïssé classes

## 1. INTRODUCTION

In this paper we describe a general 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 [10], 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 [8, 6, 4, 5, 9]. As demonstrated in [9], 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 [9], 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:

[*]Authors supported by ERC Starting Grant "Sosna"

*PODS'13,* June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2066-5/13/06 ...$15.00.

*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 generic technique shows that in this setting the above problem is decidable in ExpSpace.

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

We also show how extending slightly 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 its 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 [8, 6, 4, 5, 9]. 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 considers only finite runs and 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 all 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 [9].

## 2. DATABASE-DRIVEN SYSTEMS

We model databases as finite structures over finite schemas containing relation and function symbols. We use standard terminology from model theory (see [7] for a reference); we briefly recall the relevant notions below.

### Basic notions.

A *schema* $\Sigma$ 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 $\Sigma$ 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 *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 $\Sigma$, 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 $\varphi$ holds in the structure $\mathbb{A}$ with the valuation val for its free variables.

### Database-driven systems.

A *database-driven system* over a schema $\Sigma$ is described 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:

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

where $p, q$ are control states and $\delta$ is a quantifier-free first-order formula over the schema $\Sigma$ with free variables in the set $X \times \{\mathrm{new}, \mathrm{old}\}$. The formula $\delta$ is called the *guard* of the transition and relates the values of the registers before and after the transition.

Fix a database-driven system as described above. A *configuration* is a triple $(\mathbb{D}, q, \mathrm{val})$, where:

- $\mathbb{D}$ is a database over the schema $\Sigma$;
- $q$ is a control state;
- $\mathrm{val} : X \to \mathbb{D}$ is a valuation, which maps the registers to elements in the domain of $\mathbb{D}$.

We say that there is a *transition* between configurations $(\mathbb{D}_{\mathrm{old}}, q_{\mathrm{old}}, \mathrm{val}_{\mathrm{old}})$ and $(\mathbb{D}_{\mathrm{new}}, q_{\mathrm{new}}, \mathrm{val}_{\mathrm{new}})$, if

- $\mathbb{D}_{\mathrm{old}} = \mathbb{D}_{\mathrm{new}}$ (*transitions do not modify the database*)
- There is a transition rule $q_{\mathrm{old}} \xrightarrow{\delta} q_{\mathrm{new}}$ such that

$$\mathbb{D}_{\mathrm{old}} \models_{\mathrm{val}} \delta$$

where $\mathrm{val}(x, i) = \mathrm{val}_i(x)$ for $x \in X, i \in \{\mathrm{old}, \mathrm{new}\}$.

A *run* of the system is a sequence of configurations that begins in a configuration with an initial state, and where two consecutive configurations are connected by a transition. In this paper, we are interested in finite runs. A run is *accepting* if the control state in its last configuration is accepting. It follows from the definition that for each run, there is some database $\mathbb{D}$ that is shared by all configurations in the run. We say that the run is *driven by* $\mathbb{D}$. Note that different runs of the same system may be driven by different databases.

*Example 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 system whose accepting runs trace odd-length cycles of red nodes. The system has the following components.

- The control states are $\{start, q_0, q_1, end\}$. 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 $\delta$ is

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

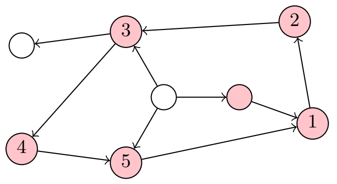
There is also a transition rule $q_1 \xrightarrow{\delta} q_0$, with the same guard. This means that the system alternates between the states $q_0$ and $q_1$, each time moving the content of register $y$ along an edge to some red node (the content of register $x$ stays in place).

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

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

There is also a transition $q_1 \xrightarrow{\alpha} end$, 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:

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

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.

*The emptiness problem.* In this paper, we study the following decision problem, which is parametrized by a class $\mathcal{C}$ of databases over a common schema $\Sigma$, and called *emptiness of database-driven systems over* $\mathcal{C}$.

- **Input.** A database-driven system over $\Sigma$.
- **Output.** Does the system have some finite 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 1. *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.*

*Existential guards.* Before describing our results in more detail, 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.

FACT 2. *For every database-driven system with existential guards one can compute in linear time a database-driven system with quantifier-free guards accepting the same runs driven by the same databases.*

However, as we shall show later on, further extensions of the guards, such as boolean combinations of existential formulas, break decidability.

## 3. DECIDABILITY RESULTS

In this section, we present the main results of the paper, which show that emptiness of database-driven systems is decidable over certain classes of databases.

## 3.1 XML documents and regular tree languages

This class is 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 sibling. 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 automaton model is presented in Section 5.3.

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 6.1. 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 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);
- 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 3. *The following problem is decidable:*

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

*For a fixed tree automaton, the problem is* PSPACE*-complete. When both the tree automaton and the system are given on input, the problem is in* EXPSPACE.

The database-driven systems are only allowed to access the trees through quantifier-free formulas that use the predicates included in TreeSchema(A). By Fact 2, we could also allow the systems 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 6) that adding any one of the above three predicates leads to undecidability.

The proof of Theorem 3 will be given in Sections 4 and 5.

*Adding data values.* We show in Section 4.4 a composition method implying that our results extend to databases storing data values allowing equality tests. In particular, the result of Theorem 3 remains valid if each node of the tree also carries a data value in $\mathbb{N}$ and the query can test these values using equalities or inequalities. The complexity bound is not affected by this extension.

## 3.2 Homomorphims

In this section, we consider schemas 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. By $\mathcal{HOM}(\mathbb{H})$ we denote the class of all databases over the schema of $\mathbb{H}$ that map homomorphically to $\mathbb{H}$. In other words, $\mathbb{G} \in \mathcal{HOM}(\mathbb{H})$ if and only if there is a homomorphism $f : \mathbb{G} \to \mathbb{H}$.

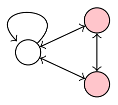
The database $\mathbb{H}$ is called the *template* for the class $\mathcal{HOM}(\mathbb{H})$. Examples of $\mathcal{HOM}(\mathbb{H})$ include $n$-colorable graphs for every $n$ (when $\mathbb{H}$ is a $n$-clique).

We show that if a class of databases can be defined as $\mathcal{HOM}(\mathbb{H})$ for some $\mathbb{H}$, then it admits an algorithm for emptiness of database-driven systems. As for Theorem 3, we actually prove a stronger result where the template is also part of the input.

THEOREM 4. *The following problem is* PSPACE*-complete.*

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

*Example 2.* 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 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 $\mathbb{G} \in \mathcal{HOM}(\mathbb{H})$ such that the system has an accepting run driven by $\mathbb{G}$.

The proof of Theorem 4 will be given in Section 4.

*Adding data values.* As for the previous case, the result of Theorem 4 remains valid if each node of the tree also carries a data value in $\mathbb{N}$ and the query can test these values using equalities or inequalities.

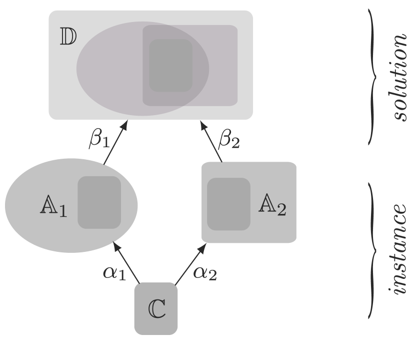
## 4. THE METHOD

Both decidability results stated in the previous section, namely Theorems 3 and 4, are proved using the same method. The method is presented in this section. The general idea is to add some more predicates or functions to the databases, so that the resulting class of databases has good closure properties, of which the most important is closure under amalgamation.

## 4.1 Fraïssé classes and 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. $\beta_1 \circ \alpha_1 = \beta_2 \circ \alpha_2$.

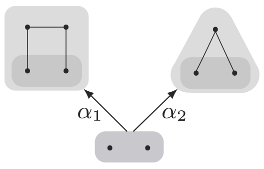
Following [7, Chapter 6, Section 6.1] a *Fraïssé class* is a class $\mathcal{C}$ of databases over a common schema such that:

– $\mathcal{C}$ is *closed under embeddings*: if $\mathbb{C}$ is a database in $\mathcal{C}$ and $\mathbb{D}$ is any database over the same schema that embed into $\mathbb{C}$, then $\mathbb{D} \in \mathcal{C}$;

– $\mathcal{C}$ is *closed under amalgamation*: every instance of amalgamation, where the databases $\mathbb{A}_1, \mathbb{A}_2, \mathbb{C}$ all belong to $\mathcal{C}$, has a solution $\mathbb{D}$ that belongs to $\mathcal{C}$; and

– $\mathcal{C}$ has the *joint embedding property*: every two databases from $\mathcal{C}$ can be embedded into a single database from $\mathcal{C}$.

*Example 3.* Consider a schema with one binary relation. The reader can verify that Fraïssé classes over this schema include: all finite linear orders, all finite directed graphs, and all equivalence relations over finite sets. The class of forests (understood as directed graphs) is not closed under amalgamation: the instance depicted below does not have a solution which is a forest.



We will show that, under weak assumptions, emptiness for database-driven systems is decidable over Fraïssé classes. The weak assumptions are that membership in the Fraïssé class is decidable, and that if the schema contains function symbols, then sets of elements of bounded size cannot generate databases of unbounded size via images of functions.

Let $\mathbb{C}$ be a database and $S$ a subset of its domain. We say that $S$ *generates* $\mathbb{C}$ if there is no proper substructure $\mathbb{C}' \subsetneq \mathbb{C}$ that contains $S$. A database is called *$n$-generated* if its domain has a subset of size $n$ that generates it. The *blowup function* of a Fraïssé class $\mathcal{C}$ is the function

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

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

THEOREM 5. *Let $\mathcal{C}$ be a Fraïssé class with membership in* PSPACE. *Emptiness of database-driven systems is decidable for $\mathcal{C}$ in space*

$$\log(n) \cdot \textit{poly}(\text{blowup}_{\mathcal{C}}(2k))$$

*where $n$ is the number of control states and $k$ is the number of registers in the database-driven system.*

PROOF. We describe a nondeterministic algorithm for the emptiness problem. A configuration $(\mathbb{A}, q, \text{val})$ of the system is called *small* if the database $\mathbb{A}$ belongs to $\mathcal{C}$ and is generated by the contents of the registers as described by val. Note that the size of $\mathbb{A}$ is bounded by $\text{blowup}_{\mathcal{C}}(k)$ and by our assumption on membership in $\mathcal{C}$, testing $\mathbb{A} \in \mathcal{C}$ requires space polynomial in $\text{blowup}_{\mathcal{C}}(k)$. Consider two small configurations

$$(\mathbb{A}_{\text{old}}, q_{\text{old}}, \text{val}_{\text{old}}) \qquad (\mathbb{A}_{\text{new}}, q_{\text{new}}, \text{val}_{\text{new}}).$$

We say that there is a *sub-transition* between them if there is a database $\mathbb{A} \in \mathcal{C}$ and two embeddings

$$f_{\text{old}} : \mathbb{A}_{\text{old}} \to \mathbb{A} \qquad f_{\text{new}} : \mathbb{A}_{\text{new}} \to \mathbb{A}$$

such that

$$(\mathbb{A}, q_{\text{old}}, f_{\text{old}} \circ \text{val}_{\text{old}}) \qquad (\mathbb{A}, q_{\text{new}}, f_{\text{new}} \circ \text{val}_{\text{new}})$$

is a transition of the system. Checking if there is a sub-transition requires space polynomial in $\text{blowup}_{\mathcal{C}}(2k)$ because the class is closed under embeddings and we can therefore assume that the database $\mathbb{A}$ is generated by the images of the valuations $f_{\text{old}} \circ \text{val}_{\text{old}}$ and $f_{\text{new}} \circ \text{val}_{\text{new}}$. This leads to the following nondeterministic algorithm.

1. Nondeterministically guess a small configuration where the state is initial.

2. If the state of the current configuration is final, then terminate and accept. Otherwise, nondeterministically guess a new small configuration accessible from the previous one by a sub-transition. Repeat step 2.

The space consumption of the algorithm is as required by the theorem. The proof of its correctness is based on the fact that the class $\mathcal{C}$ is closed under embeddings and under amalgamation. In particular, from a run of our algorithm we construct a run of the system by amalgamating the databases that appear in the small configurations.

**Completeness**. Consider an accepting run of the system, driven by some database $\mathbb{A} \in \mathcal{C}$:

$$(\mathbb{A}, q_0, \text{val}_0)(\mathbb{A}, q_1, \text{val}_1) \cdots (\mathbb{A}, q_n, \text{val}_n)$$

Let $\mathbb{B}_i$ be the substructure of $\mathbb{A}$ generated by the content of the registers at step $i$. By closure under embedding, $\mathbb{B}_i \in \mathcal{C}$

for all $i$. Moreover $\mathbb{A}$ witnesses the fact that there is a sub-transition from $(\mathbb{B}_i, q_i, \mathrm{val}_i)$ to $(\mathbb{B}_{i+1}, q_{i+1}, \mathrm{val}_{i+1})$. Hence our algorithm has an accepting run.

**Soundness**. Assume that our algorithm has a run:

$$(\mathbb{B}_0, q_0, \mathrm{val}_0)(\mathbb{B}_1, q_1, \mathrm{val}_1)\cdots(\mathbb{B}_n, q_n, \mathrm{val}_n)$$

By induction on $n$ we exhibit a database $\mathbb{B}$ and embeddings $f_i : \mathbb{B}_i \to \mathbb{B}$ such that

$$(\mathbb{B}, q_0, f_0 \circ \mathrm{val}_0)(\mathbb{B}, q_1, f_1 \circ \mathrm{val}_1)\cdots(\mathbb{B}, q_i, f_n \circ \mathrm{val}_n)$$

is a valid run of the database-driven system. For $n = 0$ we take $\mathbb{B} = \mathbb{B}_0$ and $f_0$ the identity.

Assume we have $\mathbb{B}$ for the run until step $(n-1)$ and consider $(\mathbb{B}_n, q_n, \mathrm{val}_n)$. By definition there is a database $\mathbb{A}$ and embeddings $g_{n-1} : \mathbb{B}_{n-1} \to \mathbb{A}$ and $g_n : \mathbb{B}_n \to \mathbb{A}$ such that there is a transition from $(\mathbb{A}, q_{n-1}, g_{n-1} \circ \mathrm{val}_{n-1})$ to $(\mathbb{A}, q_n, g_n \circ \mathrm{val}_n)$. But we also have an embedding $f_{n-1} : \mathbb{B}_{n-1} \to \mathbb{B}$. By closure under amalgamation, we get a database $\mathbb{A}'$ and embeddings $f : \mathbb{A} \to \mathbb{A}'$ and $g : \mathbb{B} \to \mathbb{A}'$ with good commuting properties. It is now easy to verify that $\mathbb{A}'$ is the database we where looking for with embeddings $g \circ f_i$ for $i < n$ and $f \circ g_n$ for $i = n$.

This proves the correctness of the algorithm. □

Observe that the algorithm does not use the joint embedding property. There are two (related) reasons why we use the joint embedding property: first, it is part of the classical definition of a Fraïssé class; second it is necessary for the Fraenkel-Mostowski approach described in the Section 4.5.

## 4.2 Semi-Fraïssé classes

For some of the classes we are interested in, Theorem 5 will not work, because the classes are not closed under amalgamation.

*Example 4.* Consider graphs that are 2-colorable or, equivalently, have no odd-length cycle. This class is $\mathcal{HOM}(\mathbb{H})$ where $\mathbb{H}$ is a 2-clique, over the schema $\Sigma$ 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 3.

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 $\Gamma$, 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 $\mathcal{HOM}(\tilde{\mathbb{H}})$ 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 $\Gamma$ and let $\Sigma$ be a subset of the schema $\Gamma$. The $\Sigma$-*projection* $\Sigma(\mathbb{G})$ *of* $\mathbb{G}$ is the same as $\mathbb{G}$, only the interpretation is restricted to the smaller schema. The $\Sigma$-projection of a class $\mathcal{C}$ of databases over $\Gamma$, denoted by $\Sigma(\mathcal{C})$, is defined pointwise.

The proof of the following lemma is fairly simple as quantifier-free formulas are invariant under extending the domain or the schema. For a class of databases $\mathcal{C}$, by substructures$(\mathcal{C})$ we denote the smallest class of databases containing $\mathcal{C}$ that is closed under embeddings.

Lemma 6. *Let $\mathcal{C}$ be a class of finite databases over a schema $\Sigma$. Suppose that $\mathcal{D}$ is a Fraïssé class over a schema $\Gamma \supseteq \Sigma$, such that*

$$\mathcal{C} \subseteq \Sigma(\mathcal{D}) \subseteq \mathrm{substructures}(\mathcal{C}).$$

*Then emptiness of database-driven systems over $\mathcal{C}$ is decidable with the same complexity bounds as over $\mathcal{D}$.*

A class $\mathcal{C}$ for which there exists a Fraïssé class $\mathcal{D}$ that satisfies the assumptions of the above lemma will be called a *semi-Fraïssé* class. In Example 4, the class $\mathcal{HOM}(\mathbb{H})$ is semi-Fraïssé, as witnessed by $\mathcal{HOM}(\tilde{\mathbb{H}})$.

## 4.3 HOMs are Semi-Fraïssé

As a simple application of the method, we prove Theorem 4. By Theorem 5 and Lemma 6 it is a consequence of the following lemma:

Lemma 7. *If $\mathbb{H}$ is a finite database, then $\mathcal{HOM}(\mathbb{H})$ is a semi-Fraïssé class.*

Proof. The schema $\Gamma$ is the schema $\Sigma$ 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 $\Gamma$, where a node $h \in \mathbb{H}$ gets the color $h$. It is easy to see that $\mathcal{HOM}(\mathbb{H})$ is the $\Sigma$-projection of $\mathcal{HOM}(\tilde{\mathbb{H}})$. To complete the lemma, we prove that $\mathcal{HOM}(\tilde{\mathbb{H}})$ 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 $\mathbb{D} \in \mathcal{HOM}(\tilde{\mathbb{H}})$. 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 contains no function symbols, hence we have $\mathrm{blowup}_{\mathcal{HOM}(\tilde{\mathbb{H}})}(n) = n$, and the complexity is PSpace as desired.

## 4.4 Data values

In this section we show how Theorems 3 and 4 extend to databases whose nodes are additionally equipped with data values, and where the transition systems may test equality and inequality of data values. 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.

*Homogeneous structures.* The notion of homogeneity comes from model theory. An infinite structure $\mathbb{F}$ over a schema $\Sigma$ is called *homogeneous* if every isomorphism $f : \mathbb{F}_1 \to \mathbb{F}_2$ between two finite substructures $\mathbb{F}_1, \mathbb{F}_2$ of $\mathbb{F}$ can be extended to an automorphism $\tilde{f}$ of $\mathbb{F}$.

Homogeneous structures abound; important examples include:

- The set of natural numbers, with the equality relation denoted $\sim$, denoted $\langle \mathbb{N}, \sim \rangle$;
- The rational numbers, with the linear ordering, denoted $\langle \mathbb{Q}, < \rangle$.

A theorem of Fraïssé (see [7, Chapter 6] and also Section 4.5) says that we can associate to every Fraïssé class an infinite countable structure, called the *Fraïssé limit of the Fraïssé class* which is a homogeneous structure.

*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 $\langle \mathbb{N}, \sim \rangle$ or $\langle \mathbb{Q}, < \rangle$.

Consider a finite database $\mathbb{A}$ over a schema $\Sigma$. Let $\lambda : \mathbb{A} \to \mathbb{F}$ be any labeling of the nodes 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 $\lambda$: 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 nodes 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 $\lambda$ 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 $\lambda$ is *injective*, i.e. each node gets a different data value[1].

*Example 5.* Let $t$ be a finite tree and let Treedb$(t)$ be the corresponding database. Let $\lambda \colon t \to \mathbb{N}$ be a labeling of the nodes of $t$ by natural numbers. Then the database Treedb$(t) \otimes \lambda$ can be seen as a tree equipped with data values (or *attributes*); two nodes $x, y$ store the same attribute if $x \sim y$.

*Example 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 $\mathbb{G} \otimes \lambda$ can be seen as a graph whose nodes are natural numbers: because $\lambda$ 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 whose proof is omitted here.

Proposition 1. *Let $\mathbb{F}$ be a purely relational homogeneous structure, such that deciding whether a finite database embeds into $\mathbb{F}$ can be done 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 $\mathcal{C}$.*

As a consequence of Proposition 1 and Lemma 7 we get the following extension of Theorem 4.

[1]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 5 and 6.

Corollary 8. *The following problem is decidable in* PSpace*:*

- **Input.** *A relational database $\mathbb{H}$ a database-driven system over the union of the schemas of $\mathbb{H}$ and $\mathbb{F}$.*
- **Output.** *Is there a database $(\mathbb{A} \otimes \lambda) \in \mathcal{HOM}(\mathbb{H}) \odot \mathbb{F}$ and an accepting run of the system driven by $\mathbb{A}$?*

Special cases of this result – without the condition $\mathbb{A} \in \mathcal{HOM}(\mathbb{H})$ – have been proved earlier for $\mathbb{F} = \langle \mathbb{N}, \sim \rangle$ in [5] and $\mathbb{F} = \langle \mathbb{Q}, < \rangle$ in [4].

Our abstract machinery applies also to database-driven systems, where the databases are trees with data values.

Theorem 9. *The following problem is decidable:*

- **Input.** *A tree automaton defining a language $L$ of trees labeled by an alphabet $A$, and a database-driven system over the schema* TreeSchema(A) $\cup \{\sim\}$.
- **Output.** *Is there a tree $t \in L$, a labeling $\lambda$ 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* PSpace*-complete. When both the tree automaton and the system are input, the problem is in* ExpSpace.

*Remark 1.* Theorem 9 works for any countable homogeneous structure $\mathbb{F}$ such that testing whether a given finite database $\mathbb{A}$ embeds into $\mathbb{F}$ can be done in PSpace. For instance $\langle \mathbb{N}, \sim \rangle$ could be replaced by $\langle \mathbb{Q}, < \rangle$. Actually it only matters that substructures($\mathbb{F}$) is a Fraïssé class. In particular, as

$$\text{substructures}(\langle \mathbb{N}, < \rangle) = \text{substructures}(\langle \mathbb{Q}, < \rangle),$$

the result also hold for $\mathbb{F} = \langle \mathbb{N}, < \rangle$. Similarly, by considering semi-Fraïssé classes instead of Fraïssé, the result also works with $\langle \mathbb{N}, < \rangle$ augmented with constants, thus capturing the setting of [9].

## 4.5 Fraenkel-Mostowski sets

In this section, we comment on a bigger picture that contains Theorem 5, but also implies other results, such as emptiness of database-driven systems with pushdowns. To simplify the discussion, we only focus on decidability and not on complexity. The bigger picture is called nominal sets, or Fraenkel-Mostowski sets.

*The Fraïssé limit.* We begin by observing that instead of talking about a class of finite databases, we can talk about a single limit structure (which is usually infinite so it should not be called a database). The theorem of Fraïssé says that if $\mathcal{C}$ is a Fraïssé class, then there exists a single countable (but usually infinite) homogeneous structure $\mathbb{F}$ – the *Fraïssé limit* of $\mathcal{C}$ – such that the databases in $\mathcal{C}$ are exactly the finitely generated substructures of $\mathbb{F}$. Fraïssé limits have many good model-theoretic properties, for instance they are $\omega$-categorical.

What is the connection with database-driven systems? A run of a database-driven system is finite, and therefore visits only finitely many elements of a database with its registers. It follows that a database-driven system has a run driven by some finite database in $\mathcal{C}$ if and only if it has a run driven by the Fraïssé limit of $\mathcal{C}$. Therefore, instead of studying emptiness over a class $\mathcal{C}$, we could study emptiness of a system that uses registers to store elements of the Fraïssé limit.

*Fraenkel-Mostowski sets and their automata.* Automata that store values from a Fraïssé limit in their registers have already been studied in [2], as part of a more general framework called *Fraenkel-Mostowski sets.* From the results in [2] it follows that emptiness for such automata is decidable, which implies the decidability result in Theorem 5. (We included a proof of Theorem 5 to make this paper self-contained, and also to get the precise complexity.) Apart from finite automata with registers, the Fraenkel-Mostowski framework contains other computational devices with decidable emptiness, which can then be used to get decidability results for extensions of database-driven systems. The results concerning these devices include:

- Emptiness is decidable for pushdown automata, which are allowed to store elements of a Fraïssé limit both in their state and on the pushdown, see [2]. This implies decidable emptiness for a natural pushdown extension of database-driven systems.
- Emptiness is decidable for tree automata, where a configuration can have more than one successor configuration. This implies decidable emptiness for a natural branching extension of database-driven systems.
- Under additional assumptions, which hold for regular tree languages but not for equivalence relations and HOMs, even certain alternating automata have decidable emptiness [1]. This implies decidable emptiness for a certain alternating extension of database-driven systems.

We would like to point out that the first two results (pushdown automata and tree automata) can be easily obtained without using the abstract framework of Fraenkel-Mostowski sets (this is no longer true for alternating automata, where the proof is quite involved and follows the lines of [3]). We believe, however, that seeing database-driven systems as a special case of automata in Fraenkel-Mostowski sets gives a uniform explanation for the decidability results. We plan to given a more detailed discussion of the Fraenkel-Mostowski connection, including a precise definition of the extended database-driven models, in the full version of this paper.

## 5. REGULAR TREE LANGUAGES

In this section, we prove Theorem 3, which is the main technical result of the paper. The theorem says that emptiness is decidable for database-driven systems over regular tree languages. Since the proof is quite technical, we begin by illustrating the main ideas in the case of words.

### 5.1 Regular Word Languages

Like in the case of trees, to a word $w$ over an alphabet $A$, we associate a database $\mathrm{Worddb}(w)$, where the domain is the positions of the word, there are unary predicates $\{a(x)\}_{a \in A}$ for the labels, and a binary predicate $x < y$ for the natural order on word positions. Call $\mathrm{WordSchema}(A)$ the schema of this database.

THEOREM 10. *The following problem is* PSPACE*-complete.*

- **Input.** *A regular word language $L \subseteq A^*$, given by an NFA, and a database-driven system over the schema* $\mathrm{WordSchema}(A)$.
- **Output.** *Is there a word $w \in L$ and an accepting run of the system driven by* $\mathrm{Worddb}(w)$*?*

The rest of Section 5.1 is devoted to showing the above theorem. Fix a regular word language $L \subseteq A^*$. Let $Q$ be the states of an NFA that recognizes $L$. Define

$$\mathrm{Worddb}(L) = \{\mathrm{Worddb}(w) : w \in L\}.$$

For a class of databases $\mathcal{C}$, let $\mathcal{C}^*$ denote the closure of $\mathcal{C}$ under disjoint unions. The point of studying $\mathcal{C}^*$ is that it is guaranteed to have the joint embedding property. In the specific case of $\mathcal{C} = \mathrm{Worddb}(L)$, the disjoint union is defined so that positions from different words are incomparable with respect to $<$.

We will prove that $\mathrm{Worddb}(L)^*$ is a semi-Fraïssé class, and therefore has decidable emptiness for database-driven systems. The following lemma reduces emptiness from $\mathrm{Worddb}(L)$ to $\mathrm{Worddb}(L)^*$.

LEMMA 11. *For every database-driven system $\mathcal{S}$, there is a database-driven system $\mathcal{S}^*$, such that emptiness of $\mathcal{S}$ over* $\mathrm{Worddb}(L)$ *is equivalent to emptiness of $\mathcal{S}^*$ over* $\mathrm{Worddb}(L)^*$.

PROOF SKETCH. Define the system $\mathcal{S}^*$ as extending $\mathcal{S}$ with a new register. The idea is that this new register stores some position of the word. The new register does not change contents during the whole run, and all other registers are required to be comparable with the new register in the order $<$. Apart from this, the other registers behave as in the system $\mathcal{S}$. Even when driven by a disjoint union of words, the registers will only use positions from one of the words. $\square$

From this point, our aim is to prove that $\mathrm{Worddb}(L)^*$ is a semi-Fraïssé class. In particular, we do not consider database-driven systems any more.

Fix an automaton $\mathcal{A}$ recognizing the language $L$. We assume the automaton does not contain useless states: every state in the automaton is reachable from some initial state, and that from every state an accepting state can be reached. We also assume that for each state $q$ of the automaton, there is a unique letter $a$ that can be read in that state, i.e. a unique letter such that the automaton contains transitions of the form $p \xrightarrow{a} q$ (the state $q$ is not unique, of course). This assumption can be enforced by splitting each state into one copy for each letter of the input alphabet. Denote by $\to$ the one-step reachability relation on states in the automaton, i.e. $p \to q$ holds if there is a transition $p \xrightarrow{a} q$. Let $\to^+$ be the transitive closure of this relation, i.e. reachability via nonempty words. We will be interested in strongly connected components of this relation, which we call *components.* We adopt the convention that if a state $q$ is not reachable from itself, then it is also in a component, which contains only the state $q$. Thanks to this convention, the components form a partition of the states of the automaton. We denote components by $\Gamma$.

We define a *pre-run* of the automaton to be an input word, together with a labeling of positions by states, where position $x$ gets the state after reading it. (In particular, the first state of the run, before reading any position, does not appear in the labeling.) A pre-run, call it $\rho$, is interpreted as a database, denoted by $\mathrm{Rundb}(\rho)$ as follows:

– There are the original predicates $\{a(x)\}_{a \in A}$ and $x < y$ for the input word. In other words, $\text{Rundb}(\rho)$ extends the database $\text{Worddb}(w)$, where $w$ is the input word in the run.

– There are unary state predicates $\{q(x)\}_{q \in Q}$ for the states in the run.

– For each component $\Gamma$ of the automaton, there is a unary function $\mathit{leftmost}_\Gamma(x)$ that maps a position $x$ to the leftmost position before $x$ that has a state in component $\Gamma$. If there is no such appearance, then $\mathit{leftmost}_\Gamma(x)$ is $x$. Likewise, we have a $\mathit{rightmost}_\Gamma(x)$ unary function[2]. We use the name *pointers* for the *leftmost* and *rightmost* functions.

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

$$\{\text{Rundb}(\rho) : \rho \text{ is a run.}\}.$$

Proposition 2. *$\mathcal{C}$ is closed under amalgamation.*

Before showing the proposition, we show how it implies Theorem 10.

Proof of Theorem 10. The theorem will follow from the items below thanks to Lemmas 6 and 11.

1. It is not difficult to see that the projection assumption (when projecting a pre-run to its input word) in Lemma 6 is satisfied:

$$\text{Worddb}(L)^* \subseteq \text{WordSchema}(\mathcal{C})^* \subseteq \\ \subseteq \mathit{substructures}(\text{Worddb}(L)^*)$$

2. $\mathcal{C}^*$ is a Fraïssé class. The class $\mathcal{C}^*$ is closed under substructures by definition. It is also closed under isomorphism. The joint embedding property is easy because we can simply take disjoint unions[3]. Closure of a class under amalgamation is preserved by the operation $\mathcal{C} \mapsto \mathcal{C}^*$, and therefore $\mathcal{C}^*$ is closed under amalgamation by Proposition 2. In conclusion, $\mathcal{C}^*$ is a Fraïssé class, and so $\text{Worddb}(L)^*$ is a semi-Fraïssé class.

3. The blowup of $\mathcal{C}^*$ is small. There are at most $|Q|$ components in the automaton. Since we have two unary functions per component, the blowup function for $\mathcal{C}^*$ is at most $n \mapsto 2|Q| \cdot n$. This gives the PSpace complexity bound.

□

We now resume the proof of Proposition 2. We will use the following characterization of $\mathcal{C}$.

Lemma 12. *Let $\rho$ be a pre-run, where the states are $q_1, \ldots, q_n$ listed from left to right. Then* $\text{Rundb}(\rho) \in \mathcal{C}$ *if and only if*

$$q_1 \to^+ q_2 \to^+ \cdots \to^+ q_n$$

[2]It would seem more natural to define $\mathit{leftmost}_\Gamma$ and $\mathit{rightmost}_\Gamma$ as nullary functions, i.e. constants. We choose unary functions for two reasons: to make the tree case more similar, and to make disjoint unions of runs easier.

[3]The joint embedding property is the reason why we work with $\mathcal{C}^*$ and $\text{Worddb}(L)^*$ instead of $\mathcal{C}$ and $\text{Worddb}(L)$.

Instead of proving that $\mathcal{C}$ is closed under amalgamation, we prove that it is closed under *inclusion amalgamation*. An instance of inclusion amalgamation consists of two databases $\mathbb{A}$ 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.

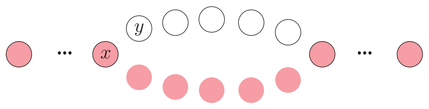
Lemma 13. *Let $\mathcal{C}$ be a class of structures closed under isomorphism. Then $\mathcal{C}$ is closed under amalgamation if and only if it is closed under inclusion amalgamation.*

Proof of Proposition 2. The proof is more wordy than it needs to be, because we want it to have the same structure as the proof for the more complicated case of trees.

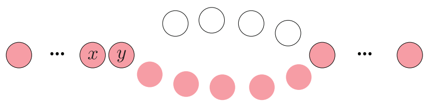
Consider an instance of inclusion amalgamation in the class $\mathcal{C}$, i.e. two pre-runs $\rho_1$ and $\rho_2$ such that the databases $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2)$ are consistent and in $\mathcal{C}$. Define $\text{Rundb}(\rho)$ to be the common part, i.e. the intersection of the two databases, which is a substructure of both, so it also belongs to $\mathcal{C}$.

We need to show a database in $\mathcal{C}$ that contains both $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2)$ as substructures. The proof is by induction on the number of elements in $\rho_1$ not in $\rho$. In the induction base $\text{Rundb}(\rho_1)$ is a substructure of $\text{Rundb}(\rho_2)$, and we already have a solution to amalgamation.

For the induction step, suppose that $y$ is in the domain of $\rho_1$, but not in $\rho$. Choose $y$ so that its preceding position in $\rho_1$, call it $x$, is already in $\rho$. The situation is illustrated in the following picture:



In the picture, the positions of $\rho_1$ are colored, the positions of $\rho_2$ have a black border, the positions of $\rho$ are colored and have a black border. To advance the induction, we add $y$ to $\rho_2$, as in the following picture:



Define a pre-run $\rho_2'$, by adding position $y$ (with its state and input label) to $\rho_2$ right after $x$, with the same state as $y$. We claim that

1. $\text{Rundb}(\rho_2') \in \mathcal{C}$;

2. $\text{Rundb}(\rho_2) \subseteq \text{Rundb}(\rho_2')$;

3. $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2')$ are consistent.

If we prove the claims above, then we are done. This is because $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2')$ are an instance of inclusion amalgamation with a smaller induction parameter. The induction assumption says that some database in $\mathcal{C}$ contains both $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2')$, and therefore it also contains $\text{Rundb}(\rho_1)$ and $\text{Rundb}(\rho_2)$.

Let $\Gamma$ be the component of the state in $y$. Consider the values of the pointers $\mathit{leftmost}_\Gamma$ that are assigned to the position $x$ in the three databases $\text{Rundb}(\rho)$, $\text{Rundb}(\rho_1)$ and

Rundb($\rho_2$). Because the databases are consistent, these are all the same position, i.e.

$$\mathit{leftmost}^{\rho}_{\Gamma}(x) = \mathit{leftmost}^{\rho_1}_{\Gamma}(x) = \mathit{leftmost}^{\rho_2}_{\Gamma}(x)$$

Call the position above $y_{\text{left}}$, it is before $y$ in $\rho_1$. Likewise, we define a position $y_{\text{right}}$.

To prove Rundb($\rho'_2$) $\in \mathcal{C}$, we use Lemma 12. By this lemma, in the run $\rho_2$, all positions between $y_{\text{left}}$ and $y_{\text{right}}$ have states in component $\Gamma$. The position $y$ is added between these positions, so it does not violate the condition from Lemma 12.

To prove Rundb($\rho_2$) $\subseteq$ Rundb($\rho'_2$), we only need to show that the functions in Rundb($\rho'_2$) are defined the same way for the positions from Rundb($\rho_2$). This is not difficult to see, because in Rundb($\rho'_2$) there is one new position, which is both followed and preceded by states in the same component. The same argument shows that Rundb($\rho_1$) and Rundb($\rho'_2$) are consistent. $\square$

## 5.2 Proof Strategy for Regular Tree Languages

We now resume the proof of Theorem 3, which says that emptiness is decidable for database-driven systems over regular tree languages. We use the same proof strategy as for words. For a tree language $L$, define

$$\text{Treedb}(L) = \{\text{Treedb}(t) : t \in L\}.$$

Like in the case of words, we will have a class $\mathcal{C}$ that represents substructures of runs.

## 5.3 Tree automata and their components

We begin by presenting the model of tree automata that we use. Out of the many equivalent models of automata on unranked trees, we choose a model where the runs are easier to pump.

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 $Q$. A *run* of the automaton over an input tree is a labeling of the tree nodes by states from $Q$, subject to some local consistency requirements described below.

– As in the word case, we assume that for each state $q$, there is a unique input letter $a$ that can be used in that state.

– The automaton has distinguished subsets of: *leaf states*, which are the only states allowed for leaves, *root states*, which are the only states allowed for the root, and *rightmost states*, which are the only states allowed for rightmost children.

– A binary first-child relation $\to_{\mathit{firstchild}}$ on states. In a run, if a node has state $q$ and its leftmost child has state $p$, then $p \to_{\mathit{firstchild}} q$ holds.

– A binary next-sibling relation $\to_{\mathit{nextsibling}}$ on states. In a run, if a node has state $q$ and its next sibling has state $p$, then $p \to_{\mathit{nextsibling}} q$ holds.

A tree is accepted by the automaton if it admits some run (we do not distinguish between runs and accepting runs). Let us fix for the rest of Section 5 a tree automaton as described above, which recognizes a tree language $L$.

*Components.* Define two binary relations on states of the automaton: a relation

$$\to_h \overset{def}{=} \to^{+}_{\mathit{nextsibling}}$$

that corresponds to following sibling, and a relation

$$\to_v \overset{def}{=} \left(\to_{\mathit{firstchild}} \circ \to^{*}_{\mathit{nextsibling}}\right)^{+}$$

that corresponds to descendant. We use the name *descendant component* for strongly connected components of the relation $\to_v$, and the name *horizontal component* for strongly connected components of the relation $\to_h$. Again, we adopt the convention that when a state is not reachable from itself by $\to_v$, then it still forms a (singleton) descendant component, likewise for horizontal component. We distinguish two kinds of descendant components:

– A descendant component $\Gamma$ is called *branching* if in some run, some node with state in $\Gamma$ has two children with states in $\Gamma$.

– A descendant component $\Gamma$ is called *linear* otherwise. This means that in every run, every node with state in $\Gamma$ has at most one child with a state in $\Gamma$.

For a descendant component, define a set of states $\mathit{left}(\Gamma)$ as follows. Suppose that in some run, a node $x$ has two descendants $y$ and $z$, such that $z$ is before $y$ in document order, and is not on the path from $x$ to $y$. If the states in $x$ and $y$ are in $\Gamma$, then we put the state in $z$ into the set $\mathit{left}(\Gamma)$. The set $\mathit{right}(\Gamma)$ is defined similarly.

## 5.4 The class $\mathcal{C}$

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

*The pointers.* Define a *pre-run* to be any tree where each node is labeled by a letter $a \in A$, as well as a state $q \in Q$ such that $a$ is the unique letter that can be read in state $q$. A pre-run need not satisfy the consistency conditions in the definition of a tree automaton run. A node $x$ in a pre-run is called *component maximal* if none of its children have a state in the same descendant component. For a pre-run $\rho$, we define Rundb($\rho$) to be the following database.

– We have the standard database Treedb($t$) for the input tree $t$: the node labels, the descendant order, the document order, and the closest common ancestor function.

– For each state $q$, there is a unary function $\mathit{leftmost}_q(x)$ defined as follows. If $x$ is a component maximal node, then $\mathit{leftmost}_q(x)$ maps $x$ to the leftmost child with a state in $q$. If $x$ is not component maximal, or it has no children with state $q$, then the function is "undefined", which is encoded by $\mathit{leftmost}_q(x) = x$.

– In the same way, we define a function $\mathit{rightmost}_q(x)$, but for the rightmost child with a state in $\Gamma$.

– For each descendant component $\Gamma$, there is a unary function $\mathit{ancestormost}_{\Gamma}(x)$, whose value is the last node on the path from $x$ to the root that has label $q$. If there is no such node, then the function is "undefined", which is encoded by $\mathit{ancestormost}_{\Gamma}(x) = x$.

– Suppose that $x$ is a node whose state is in a linear descendant component $\Gamma$. Then $\mathit{descendantmost}(x)$ maps $x$

to the unique descendant of $x$ that has a state in $\Gamma$, and has no children with states in $\Gamma$. If the state of $x$ is not in a linear descendant component, then the function is "undefined", which is encoded by $down(x) = x$.

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

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

In other words, a database belongs to $\mathcal{C}$ if it can be extracted from a run (not a pre-run) so that nodes are extracted together with the values of their pointers. Following the same proof as in Theorem 10, to prove Theorem 3, it will be sufficient to prove the following results, whose proofs are omitted here.

LEMMA 14. *The blowup function for $\mathcal{C}^*$ is $n \mapsto c{\cdot}n$, where the constant $c$ is exponential in the state space $Q$.*

PROPOSITION 3. *$\mathcal{C}$ is closed under amalgamation.*

## 6. 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.

### 6.1 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 $succ(x, y)$ holds if $y - x = 1$.

FACT 15. *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 system 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 15 that in the presence of the *child* or *next sibling* axis it is undecidable whether a database-driven has an accepting run.

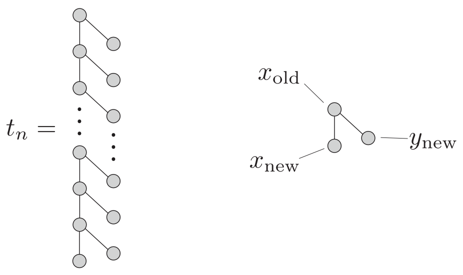
*The sibling axis.*

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

- **Input.** *A database-driven system 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:

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

These conditions guarantee that $x_{\text{new}}$ is a child of $x_{\text{old}}$ (they do not guarantee that $x_{\text{new}}$ 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 15, 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 2.* 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.

### 6.2 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 defining the transitions of the system, for instance by allowing first-order formulas, while preserving the decidability of the emptiness problem.

We have seen in Fact 2 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 15. Indeed, using boolean combinations of existential formulas, one can define the *child* axis:

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

## 6.3 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 DataTreeSchema(A) and has rules guarded by boolean combinations of formulas of the following form (called tree pattern formulas):

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

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 $\phi$ is a conjunction of conjuncts of the following three possible forms:

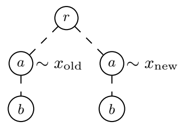
$$v_i \sim x_j, \qquad v_i \preceq_v v_j, \qquad \lambda(v_i).$$

Note that the restriction on $\phi$ implies that a formula $\delta$ 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 7.* Consider trees labeled by $\{a, b, r\}$. The following tree pattern:

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

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



THEOREM 17. *There is a language of A-labeled trees L such that the following decision problem is undecidable.*

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

Having described the applications and limitations of our framework, we end this paper.

## 7. REFERENCES

[1] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *Symp. on Principles of Programming Languages (POPL)*, pages 401–412, 2012.

[2] Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata with group actions. In *Symp. on Logic in Computer Science (LICS)*, pages 355–364, 2011.

[3] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. In *Symp. on Logic in Computer Science (LICS)*, pages 17–26, 2006.

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

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

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

[7] W. Hodges. *A shorter model theory.* Cambridge Univerity Press, 1997.

[8] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

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

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

## APPENDIX

## A. PROOF OF LEMMA 1

PROOF. 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. $\square$

## B. PROOF OF LEMMA 6

Consider a database-driven system $\mathcal{S}$ over $\mathcal{C}$.

Because $\mathcal{C} \subseteq \Sigma(\mathcal{D})$, if $\mathcal{S}$ has a run driven by a database in $\mathcal{C}$ then it has a run driven by a database in $\Sigma(\mathcal{D})$.

But adding extra relations to the database does not affect the run of the system. Hence $\mathcal{S}$ has a run driven by a database in $\Sigma(\mathcal{D})$ iff it has a run driven by a database in $\mathcal{D}$.

As $\Sigma(\mathcal{D}) \subseteq$ substructures($\mathcal{C}$) if $\mathcal{S}$ has a run driven by a database in $\Sigma(\mathcal{D})$ then it has a run driven by a database in substructures($\mathcal{C}$).

But as use only quantifier-free guards, expanding the database does not affect the run: $\mathcal{S}$ has a run driven by a database in substructures($\mathcal{C}$) iff it has a run driven by a database in $\mathcal{C}$.

Hence it is enough to test whether $\mathcal{S}$ has a run driven by a database in $\mathcal{D}$ which is decidable by Theorem 5 as $\mathcal{D}$ is Fraïssé.

## C. THE AMALGAMATION ALGORITHM

In this section, we prove the correctness of the algorithm presented Section 4, in the course of the proof of Theorem 5.

PROOF. **Completeness**. Consider an accepting run of the system, driven by some database $\mathbb{A} \in \mathcal{C}$:

$$(\mathbb{A}, q_0, \mathrm{val}_0)(\mathbb{A}, q_1, \mathrm{val}_1) \cdots (\mathbb{A}, q_n, \mathrm{val}_n)$$

Let $\mathbb{B}_i$ be the substructure of $\mathbb{A}$ generated by the content of the registers at step $i$. By closure under embedding, $\mathbb{B}_i \in \mathcal{C}$ for all $i$. Moreover $\mathbb{A}$ witnesses the fact that there is a subtransition from $(\mathbb{B}_i, q_i, \mathrm{val}_i)$ to $(\mathbb{B}_{i+1}, q_{i+1}, \mathrm{val}_{i+1})$. Hence our algorithm has an accepting run.

**Soundness**. Assume that our algorithm has a run:

$$(\mathbb{B}_0, q_0, \mathrm{val}_0)(\mathbb{B}_1, q_1, \mathrm{val}_1) \cdots (\mathbb{B}_n, q_n, \mathrm{val}_n)$$

By induction on $n$ we exhibit a database $\mathbb{B}$ and embeddings $f_i : \mathbb{B}_i \to \mathbb{B}$ such that

$$(\mathbb{B}, q_0, f_0 \circ \mathrm{val}_0)(\mathbb{B}, q_1, f_1 \circ \mathrm{val}_1) \cdots (\mathbb{B}, q_i, f_n \circ \mathrm{val}_n)$$

is a valid run of the database-driven system. For $n = 0$ we take $\mathbb{B} = \mathbb{B}_0$ and $f_0$ the identity.

Assume we have $\mathbb{B}$ for the run until step $(n-1)$ and consider $(\mathbb{B}_n, q_n, \mathrm{val}_n)$. By definition there is a database $\mathbb{A}$ and embeddings $g_{n-1} : \mathbb{B}_{n-1} \to \mathbb{A}$ and $g_n : \mathbb{B}_n \to \mathbb{A}$ such that there is a transition from $(\mathbb{A}, q_{n-1}, g_{n-1} \circ \mathrm{val}_{n-1})$ to $(\mathbb{A}, q_n, g_n \circ \mathrm{val}_n)$. But we also have an embedding $f_{n-1} : \mathbb{B}_{n-1} \to \mathbb{B}$. By closure under amalgamation, we get a database $\mathbb{A}'$ and embeddings $f : \mathbb{A} \to \mathbb{A}'$ and $g : \mathbb{B} \to \mathbb{A}'$ with good commuting properties. It is now easy to verify that $\mathbb{A}'$ is the database we where looking for with embeddings $g \circ f_i$ for $i < n$ and $f \circ g_n$ for $i = n$.

This proves the correctness of the algorithm. $\square$

## D. PROOF OF PROPOSITION 1

In this appendix, we prove Proposition 1. We start by defining some auxiliary notions.

If $\mathbb{A}$ and $\mathbb{B}$ are structures over the same schema, then we write

$$\mathbb{A} \hookrightarrow \mathbb{B}$$

if the domain of $\mathbb{A}$ is contained in the domain of $\mathbb{B}$, and the inclusion mapping is an embedding of structures. The following lemma – in fact a restatement of Lemma 13 – shows that in the definition of a Fraïssé class, one could equally well restrict to inclusions.

LEMMA 18. *Let $\mathcal{C}$ be a class of structures which is closed under isomorphisms. Then, the following conditions are equivalent.*

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

– *For any amalgamation instance in $\mathcal{C}$*

$$\mathbb{C} \hookrightarrow \mathbb{A}_1 \qquad \mathbb{C} \hookrightarrow \mathbb{A}_2, \tag{1}$$

*there exists a solution*

$$\mathbb{A}_1 \hookrightarrow \mathbb{C} \qquad \mathbb{A}_2 \hookrightarrow \mathbb{C}. \tag{2}$$

PROOF (SKETCH). This follows from the fact that Fraïssé classes are closed under isomorphisms, so any embedding $\alpha\colon \mathbb{A} \to \mathbb{B}$ is equivalent (up to isomorphism) to an inclusion $\mathbb{A}' \hookrightarrow \mathbb{B}'$. $\square$

We now turn to the proof of Proposition 1. 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 $\Sigma$. Assuming that $\mathbb{F}$ is nonempty, $\mathcal{F}$ contains at least one structure with a nonempty domain (the case of Proposition 1 when $\mathbb{F}$ is empty is easy).

LEMMA 19. *Consider an amalgamation instance in $\mathcal{F}$*

$$\mathbb{C} \hookrightarrow \mathbb{A}_1 \qquad \mathbb{C} \hookrightarrow \mathbb{A}_2, \tag{3}$$

*in which the mappings are inclusions. Let $D$ be a finite set such that*

$$|\mathbb{A}_1| \cup |\mathbb{A}_2| \subseteq D.$$

*Then the instance* (3) *has a solution $\mathbb{D}$, where*

$$D = |\mathbb{D}|,$$

*and the solution consists of inclusions:*

$$\mathbb{A}_1 \hookrightarrow \mathbb{D} \qquad \mathbb{A}_2 \hookrightarrow \mathbb{D}.$$

PROOF (SKETCH). The proof of the lemma uses the fact that in relational structures, any subset induces a substructure, so one can easily restrict a solution to the amalgamation instance (3) to one whose domain is equal to $|\mathbb{A}_1| \cup |\mathbb{A}_2|$. Further, one can extend any structure in $\mathcal{F}$ to a structure in $\mathcal{F}$ whose domain contains some additional elements. $\square$

We now continue with the proof of Proposition 1. Let $\Sigma$ 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 $\Sigma$ and $T$ are disjoint.

By $\mathcal{C} \odot \mathcal{F}$ we denote the class of structures over the union schema $\Sigma \cup T$, whose $\Sigma$-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 $\Sigma$ 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 an amalgamation instance

$$\mathbb{C} \hookrightarrow \mathbb{A}_1 \qquad \mathbb{C} \hookrightarrow \mathbb{A}_2 \tag{4}$$

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

$$\Sigma(\mathbb{C}) \hookrightarrow \Sigma(\mathbb{A}_1) \qquad \Sigma(\mathbb{C}) \hookrightarrow \Sigma(\mathbb{A}_2) \tag{5}$$

$$T(\mathbb{C}) \hookrightarrow T(\mathbb{A}_1) \qquad T(\mathbb{C}) \hookrightarrow T(\mathbb{A}_2). \tag{6}$$

Since $\mathcal{C}$ is a Fraïssé class, by Lemma 18 there exists a solution to the second instance (6),

$$T(\mathbb{A}_1) \hookrightarrow \mathbb{D}_T \qquad T(\mathbb{A}_2) \hookrightarrow \mathbb{D}_T \tag{7}$$

We now apply Lemma 19 to the first instance (5), and to $D = |\mathbb{D}_T|$, yielding a solution

$$\Sigma(\mathbb{A}_1) \hookrightarrow \mathbb{D}_\Sigma \qquad \Sigma(\mathbb{A}_1) \hookrightarrow \mathbb{D}_\Sigma \tag{8}$$

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 (7) and (8) that

$$\mathbb{A}_1 \hookrightarrow \mathbb{D} \qquad \mathbb{A}_2 \hookrightarrow \mathbb{D}$$

is a solution to (4).

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 $n \in \mathbb{N}$,

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

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 $n \in \mathbb{N}$,

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

□

## E. AMALGAMATION OF TREES

In this part of the Appendix, we give the missing proofs for results from Section 5.

### E.1 Proof of Lemma 14

In this section, we show Lemma 14, which says that the blowup function for $\mathcal{C}^*$ is $n \mapsto c \cdot n$, where the constant $c$ is exponential in the state space $Q$. It is easy to see that the classes $\mathcal{C}^*$ and $\mathcal{C}$ have the same blowup functions. Therefore, we concentrate on the blowup function for $\mathcal{C}$

Let $F$ be the functions available in the schema, i.e. $F$ is the closest common ancestor function *cca*, as well as the pointer functions

$$\textit{leftmost}_q, \textit{rightmost}_q, \textit{descendantmost}, \textit{ancestormost}_\Gamma$$

for the possible choices of $q$ and $\Gamma$. Let us write $P \subseteq F$ for the pointer functions, and partition $P$ into two parts: the part $P_\uparrow$ which contains the *ancestormost*$_\Gamma$ pointer functions, and the part $P_\downarrow$ which contains the remaining pointer functions.

Fix a pre-run $\rho$. For a set $X$ of nodes in $\rho$ and a subset $G \subseteq F$, we write $[X]^G$ for the set 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 lemmas which show that there is a natural order in applying the functions. The proofs are straightforward.

Lemma 20. $[X]^F = [[X]^{cca}]^P$

Lemma 21. $[X]^P = [[X]^{P_\uparrow}]^{P_\downarrow}$.

Combining the two lemmas, 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 a set grow by at most a multiplicative factor of the number of descendant components. Finally, closing a set under $P_\downarrow$ makes a set 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.

### E.2 Proof of Proposition 3

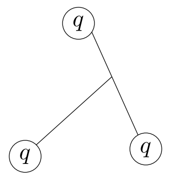
In this section, we prove Proposition 3, which says that $\mathcal{C}$ is closed under amalgamation.

#### E.2.1 A local characterization

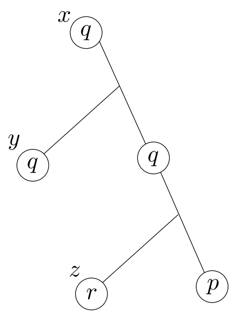
The goal of Section E.2.1 is to prove Lemma 23, which gives a local characterization of $\mathcal{C}$. We begin with an observation about branching components.

Lemma 22. *When $\Gamma$ is a branching descendant component, then left$(\Gamma)$ = right$(\Gamma)$.*

Proof. Suppose that $r \in \textit{left}(\Gamma)$, which is witnessed by a run as in the definition of *left*$(\Gamma)$. Because the vertical component $\Gamma$ is branching, there exists a run which contains the following pattern:



By combining the two runs, we get a run which contains the following pattern:



which proves that $r \in \mathit{right}(\Gamma)$. $\square$

Suppose that $x, y$ are nodes of a tree, such that the tree is implicit from the context, and therefore also the states labelling $x$ and $y$, call them $p$ and $q$, are implicit from the context. Then we write $x \to_v y$ instead of $p \to_v q$, and likewise for the other relations on states in the automaton, such as

$$\to_v^+ \qquad \to_h \qquad \to_h^+ \qquad \to_{\mathit{leftmost}} .$$

We now present the local characterization of $\mathcal{C}$.

Lemma 23. *Let $\rho$ be a pre-run. Then* $\mathrm{Rundb}(\rho)$ *belongs to $\mathcal{C}$ if and only if the root has a root state, and every node $x$ of $\rho$ satisfies the following condition.*

**Condition (*)** *Let $x_1, \ldots, x_n$ be the states in the children of $x$, listed in document order. Then*

1. *If $x$ is a leaf, then it has a leaf state.*
2. *If $x$ is a component maximal node, then*
$$x \to_{\mathit{leftmost}} x_1 \to_h^+ x_2 \to_h^+ x_3 \to_h^+ \cdots \to_h^+ x_n.$$
3. *Suppose that $x$ is not component maximal, and its state belongs to a linear component $\Gamma$. Then for some $i \in \{1, \ldots, n\}$:*
   - *The states in $x_1, \ldots, x_{i-1}$ are in left$(\Gamma)$.*
   - *The state in $x_i$ is in $\Gamma$.*
   - *The states in $x_{i+1}, \ldots, x_n$ are in right$(\Gamma)$.*
4. *Suppose that $x$ is not component maximal, and its state belongs to a branching component $\Gamma$. Then $x \to_v x_i$ holds for every $i \in \{1, \ldots, n\}$.*

Proof. By definition of components, using Lemma 22 for item 4. $\square$

*Amalgamation.* The following lemma shows that, the notion of a component maximal node is preserved in substructures.

Lemma 24. *Let $\rho$ be a pre-run. Let* $\mathrm{Rundb}(\tau)$ *be a substructure of* $\mathrm{Rundb}(\rho)$*, and let $x$ be a node in* $\mathrm{Rundb}(\tau)$*. Then $x$ is component maximal in* $\mathrm{Rundb}(\tau)$ *if and only if it is component maximal in* $\mathrm{Rundb}(\rho)$*.*

Proof. A node $x$ is component maximal if and only if either:

- The state in $x$ is a leaf state; or
- For every state $q$, the pointers $\mathit{leftmost}_q(x)$ and $\mathit{rightmost}_q(x)$ point to $x$.

This information is preserved in substructures. $\square$

Let $\tau$ and $\rho$ be pre-runs. We say that $\tau$ is a sub-run of $\rho$ if the nodes of $\tau$ are a subset of the nodes of $\rho$, and the two pre-runs agree on the labels and states, the root node, the closest common ancestor function, the document order, and the descendant order. This almost means that

$$\mathrm{Rundb}(\tau) \subseteq \mathrm{Rundb}(\rho),$$

the only thing missing is that the pointers

$$\mathit{leftmost}_q \quad \mathit{rightmost}_q \quad \mathit{ancestormost} \quad \mathit{descendantmost}$$

might be defined differently in $\rho$ and $\tau$. Lemma 25 gives sufficient conditions for the pointers to be defined the same way.

Before proving Lemma 25, we introduce some terminology. When $\tau$ is a sub-run of $\rho$, the common nodes agree on the descendant, document order, closest common ancestor, but not necessarily for sibling, next sibling, previous sibling, parent and child. To avoid confusion, we will talk about the $\tau$-parent of a node, or the $\rho$-parent, and likewise for the other terminology of tree structure.

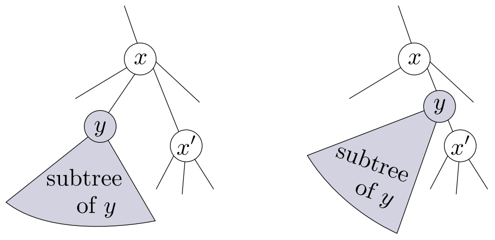


**Figure 1: The pre-runs $\tau$ and $\rho$ in Lemma 25. The node $x$ is the parent of $y$ and the node $x'$ is the node of $\rho$ that is after $y$ in document order. The left picture is when $y$ has no $\rho$-descendants in $\tau$, and the right picture is when $y$ has $\rho$-descendants. In the right picture, there can only be one $\rho$-child from $\tau$, namely $x'$, since otherwise $y$ would be a closest common ancestor of two nodes from $\tau$.**

Lemma 25. *Let $\tau$ be a pre-run that is a sub-run of a pre-run $\rho$, such that both* $\mathrm{Rundb}(\tau), \mathrm{Rundb}(\rho) \in \mathcal{C}$*. Assume further that the difference $\rho - \tau$ contains exactly node $y$ with a $\rho$-parent in $\tau$, call this $\rho$-parent $x$. (See Figure 1.)*

*To prove* $\mathrm{Rundb}(\rho) \subseteq \mathrm{Rundb}(\tau)$*, it is sufficient to show the following implications:*

1. *If $x$ is a maximal component node in $\tau$, then:*
   - *$x$ is also a maximal component node in $\rho$; and*
   - *$y$ has both a $\rho$-preceding sibling and a $\rho$-following sibling with the same state.*
2. *If $y$ has $\rho$-children in $\tau$ then $x, y$ have states in the same descendant component.*

Proof. How can $\mathrm{Rundb}(\rho) \subseteq \mathrm{Rundb}(\tau)$ fail? This can happen if the pointer functions are defined differently in $\mathrm{Rundb}(\rho)$ than in $\mathrm{Rundb}(\tau)$. Let us deal with the pointer functions one by one.

– Consider first the *leftmost*$_q$ functions. Except for $x$, which is the $\rho$-parent of $y$, all nodes in $\tau$ have the same children in $\rho$ as they have in $\tau$. For the node $x$, the value $\mathit{leftmost}_q(x)$ is the same in $\tau$ and $\rho$ because of the assumption.

– The same argument works for *rightmost*$_q$.

– Consider the $\mathit{ancestormost}_\Gamma$ functions. If $y$ is has no $\rho$-children from $\tau$, then all nodes in $\tau$ keep the same ancestors in the pre-run $\rho$, and therefore have the same values for $\mathit{ancestormost}_\Gamma$. Suppose than that $y$ has $\rho$-children in $\tau$. By the second implication, the states of $x, y$ are in the same descendant component. This means that the values of the $\mathit{ancestormost}_\Gamma$ will not be affected, because no pointer will be redirected to $y$.

– Finally, consider the $\mathit{descendantmost}_\Gamma$ function, for some linear descendant component $\Gamma$. We need to show that

$$\mathit{descendantmost}^\tau_\Gamma(z) \neq \mathit{descendantmost}^\rho_\Gamma(z) \qquad (1)$$

holds for every node $z$ from $\tau$. We only need to check the case when $z$ has a state in $\Gamma$, otherwise both pointers in (1) are self-loops. It is easy to see that (1) holds when $z$ is not a $\rho$-ancestor of $y$ in $\rho$, or when $z$ has a state in a different descendant component than $y$. Assume then that $z$ is a $\rho$-ancestor of $y$, and both $y$ and $z$ have states in component $\Gamma$. Consider the two cases as in Figure 1:

  – The first case is when $y$ has no $\rho$-children in $\tau$, which is the first picture in Figure 1. If $x$ is a component maximal node in $\tau$, then by the first implication in the assumption of the lemma, it is also a component maximal node in $\rho$, and therefore both pointers in (1) are self-loops. If $x$ is not a component maximal node in $\tau$, then $y$ cannot have a state in $\Gamma$, since otherwise $x$ would have two $\rho$-children in $\Gamma$, contradicting the assumption that $\mathrm{Rundb}(\rho) \in \mathcal{C}$.

  – The second case is when $y$ has a $\rho$-child in $\tau$, call it $x'$, which is the second picture in Figure 1. Observe that $y$ cannot have any other $\rho$-children from $\tau$, since otherwise $y$ would be a closest common ancestor of two nodes from $\tau$. By the second implication in the assumption of the lemma, the states of $x$ and $y$ are in in $\Gamma$. This means that $x$ is not component maximal in $\rho$, and therefore it cannot be component maximal in $\tau$ by the first implication in the assumption of the lemma. Therefore, some $\tau$-child of $x$ has a state in $\Gamma$. This child must be $x'$, as shown in the caption of Figure 1. Therefore, both pointers in (1) point to $x'$ or some descendant of $x'$.

□

We are now ready to prove that $\mathcal{C}$ is closed under amalgamation. By Lemma 13, 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 pre-runs $\rho_1$ and $\rho_2$ such that the databases $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho_2)$ are consistent and in $\mathcal{C}$. Define $\mathrm{Rundb}(\rho)$ to be the common part, i.e. the intersection of the two databases, which is a substructure of both, so it also belongs to $\mathcal{C}$.

We need to show a database in $\mathcal{C}$ that contains both $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho_2)$ as substructures. The proof is by induction on the number of elements in $\rho_1$ but not in $\rho$. In the induction base, $\mathrm{Rundb}(\rho_1)$ is a substructure of $\mathrm{Rundb}(\rho_2)$, and we already have a solution to amalgamation.

For the induction step, suppose that node $y$ is in $\rho_1$, but not in $\rho$. Choose $y$ so that its $\rho_1$-parent, call it $x$, is already in $\rho$. Let $x_1, \ldots, x_n$ be the $\rho$-children of $x$, listed in document order.

Choose $i \in \{1, \ldots, n+1\}$ so that $x_i$ is the first node in $\rho$ after $y$ in $\rho_1$-document order. (We adopt the convention that $i = n+1$ means that $y$ is after $x_n$). Depending on whether $y$ is an ancestor of $x_i$ or not, we get one of the two situations illustrated in Figures 2 and 3.

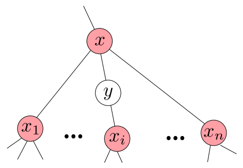


**Figure 2:** $y$ **is an ancestor of** $x_i$

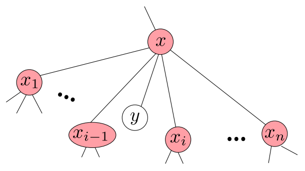


**Figure 3:** $y$ **is not an ancestor of** $x_i$

Define a pre-run $\rho'_2$ as follows:

– **Case of Figure 2.** If $y$ is an ancestor of $x_i$, then add $y$, but not the subtree of $y$, to $\rho_2$ so that $y$ becomes the first node on the path from $x$ to $x_i$.

– **Case of Figure 3.** If $y$ is not an ancestor of $x_i$, then add $y$, together with its subtree, to $\rho_2$ so that $y$ becomes the previous sibling of $x_{i-1}$. (When $i = n+1$, $y$ and its subtree are added so that $y$ is the last child of $x$.)

To complete the induction step, we need to show that

1. $\mathrm{Rundb}(\rho'_2) \in \mathcal{C}$;
2. $\mathrm{Rundb}(\rho_2) \subseteq \mathrm{Rundb}(\rho'_2)$;
3. $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho'_2)$ are consistent.

If we prove the claims above, then we are done. This is because $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho'_2)$ are an instance of inclusion amalgamation with a smaller induction parameter. The induction assumption says that some database in $\mathcal{C}$ contains both $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho'_2)$, and therefore it also contains $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho_2)$.

LEMMA 26. *In case of Figure 2, the nodes $x, y$ and $x_i$ have states in the same descendant component.*

PROOF. Let $\Gamma$ be the descendant component of the state in $y$. If $\Gamma$ does not contain the state in $x$, then

$$\mathit{ancestormost}^\rho_\Gamma(x_i)$$

would need to point a node between $x$ and $x_i$, but there is no such in $\rho$. It follows that $\square$

To prove the three items 1,2,3, we do a case distinction on the node $x$.

– $x$ is not a maximal component node in $\rho$, and the component of $x$ is branching.
  1. To prove $\mathrm{Rundb}(\rho_2') \in \mathcal{C}$, we show that every node in $\rho_2'$ satisfies condition (*) from Lemma 23. We consider three kinds of nodes: the node $y$, the node $x$, and the remaining nodes.
     – Condition (*) is satisfied for $y$. If $\rho_2'$ was constructed according to the case in Figure 3, then $y$ has the same children in $\rho_2'$ as it had in $\rho_1$, and condition (*) was satisfied for $y$ in $\rho_1$. If $\rho_2'$ was constructed according to the case in Figure 2, then $y$ has only one child in $\rho_2'$, namely the first node, call it $z$, which appears in the tree $\rho_2$ on the path from $x$ to $x_i$. By Lemma 26, the node $z$ has a label in the same descendant component as $y$, and therefore condition (*) holds for $y$.
     – Condition (*) is satisfied for $x$. Since the state of $x$ is in a branching component, condition (*) is very easy to satisfy: every $\rho_2'$-child of $x$, call it $z$, must satisfy $x \to_v z$. This holds, because every $\rho_2'$-child of $x$ is either a $\rho_1$-child of $x$, or a $\rho_2$-child of $x$.
     – Condition (*) is satisfied for the remaining nodes $y$. For these nodes, their sequence of children is either the same as in $\rho_1$ (for descendants of $y$) or the same as in $\rho_1$ (the remaining nodes). In both cases, condition (*) is satisfied.
  2. To prove $\mathrm{Rundb}(\rho_2) \subseteq \mathrm{Rundb}(\rho_2')$, we use Lemma 25. The first implication in the lemma is vacuously satisfied, because its assumption fails. The second implication is satisfied thanks to Lemma 26.
  3. To prove that $\mathrm{Rundb}(\rho_1)$ and $\mathrm{Rundb}(\rho_2')$ are consistent, we need to that they agree on their common part, call this common part $\rho'$:
$$\mathrm{Rundb}(\rho') \subseteq \mathrm{Rundb}(\rho_1)$$
$$\mathrm{Rundb}(\rho') \subseteq \mathrm{Rundb}(\rho_2').$$
     This is proved the same way as item 2.

– $x$ is not a maximal component node in $\rho$, and the descendant component of $x$ is linear. Call this component $\Gamma$.
  1. To prove $\mathrm{Rundb}(\rho_2') \in \mathcal{C}$, we show that every node in $\rho_2'$ satisfies condition (*) from Lemma 23. The only interesting case is for node $x$ (the others are shown the same way as previously.) Since the descendant component of $x$ is linear, and $x$ is not a maximal component node in $\rho$, then
$$\mathit{descendantmost}^{\rho}(x)$$
     is an $\rho$-descendant of $x$. It follows that some $\rho$-child of $x$ is an ancestor of, or equal to, the node above, and therefore has a state in component $\Gamma$. Since $x_1, \ldots, x_n$ are all the $\rho$-children of $x$, it follows that some node $x_j \in \{x_1, \ldots, x_n\}$ has a state in $\Gamma$. By Lemma 23 applied to $\rho_1$, we know that if $j \in \{1, \ldots, i-1\}$ then the state of $y$ is in $\mathit{right}(\Gamma)$, and if $j \in \{i, \ldots, n\}$ then the state of $y$ is in $\mathit{left}(\Gamma)$. In either case, the condition of Lemma 23 holds for node $x$ in $\rho_2'$.
  2,3. Here the argument is the same as in the branching case.

– $x$ is a maximal component node in $\rho$.
  1. To prove $\mathrm{Rundb}(\rho_2') \in \mathcal{C}$, we show that every node in $\rho_2'$ satisfies condition (*) from Lemma 23. The only interesting case is for node $x$ (the others are shown the same way as previously.) The only difference between the children of $x$ in $\rho_2$ and in $\rho_2'$ is that in $\rho_2'$, the node $y$ is inserted. Let $y_{prev}$ be the previous sibling of $y$ in $\rho_2'$, and let $y_{next}$ be the next sibling of $y$ in $\rho_2'$. To prove condition (*), we need to show
$$y_{prev} \to_h^+ y \to_h^+ y_{next}. \tag{2}$$
     LEMMA 27. *There are*
$$j \in \{1, \ldots, i-1\} \qquad \textit{and} \qquad k \in \{i, \ldots, n\}$$
     *such that $x_j$ and $x_k$ have the same state as $y$.*

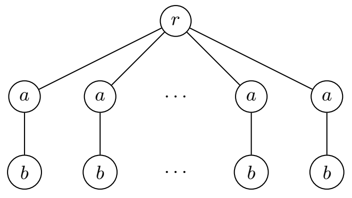
     PROOF. Let $q$ be the state in $y$. We only show the existence of $j$, the proof for $k$ is the same. The node $\mathit{leftmost}_q^{\rho_1}(x)$ must be a preceding sibling of $y$ in $\rho_1$. Because $\mathrm{Rundb}(s) \subseteq \mathrm{Rundb}(\rho_1)$, and $x$ belongs to $\rho$, we have
$$\mathit{leftmost}_q^{\rho}(x) = \mathit{leftmost}_q^{\rho_1}(x). \tag{3}$$
     The node on the left side of the equality is one of the nodes $x_1, \ldots, x_{i-1}$. $\square$

     The node $x_j$ is before $y_{next}$ in $\rho_2$, and therefore by Lemma 23 applied to the tree $\rho_2$, we have
$$x_j \to_h^+ y_{next}.$$
     Because $x_j$ has the same state as $y$, we have
$$y \to_h^+ y_{next}.$$
     In a similar way we show
$$y_{prev} \to_h^+ y.$$
     This completes the proof of (2).
  2. To prove $\mathrm{Rundb}(\rho_2) \subseteq \mathrm{Rundb}(\rho_2')$, we use Lemma 25. For the first implication in the lemma, we need to show that $y$ has both a preceding sibling and a following sibling in $\rho_2$ with the same label. This follows from Lemma 27. The second implication is satisfied thanks to Lemma 26.
  3. This is proved the same was as item 3.

## F. TREE PATTERNS

In this section, we prove Theorem 17 – that using tree patterns leads to undecidability.

PROOF PROOF OF THEOREM 17. The language $L$ consists of trees of the following form.



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 system $S_M$ 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 $t \in L$.

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 $t \in L$. Let $t_1, t_2, \ldots, t_n$ be all the maximal subtrees of $t$ not containing the root of $t$ (these trees have roots labeled by $a$).

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.

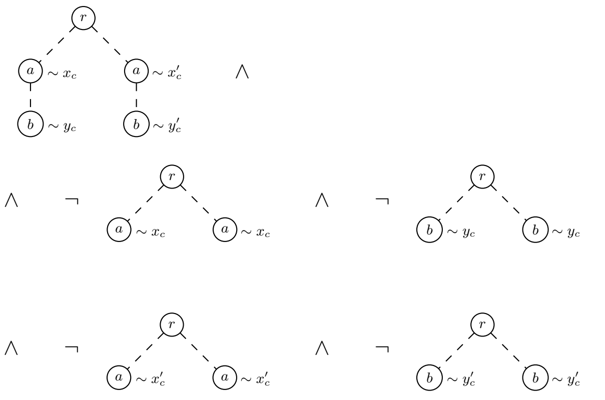
For each counter $c \in C$, the system $S_M$ will store in its variables, at any moment, two nodes $x_c, y_c$. The invariant maintained during the run is:

– 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 $t_j$. 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 7):



Satisfaction of the above formula 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. $\square$