Querying schemas with access restrictions

Michael Benedikt
Oxford University, UK
michael.benedikt@cs.ox.ac.uk

Pierre Bourhis
Oxford University, UK
pierre.bourhis@cs.ox.ac.uk

Clemens Ley
Oxford University, UK
clemens.ley@cs.ox.ac.uk

ABSTRACT

We study verification of systems whose transitions consist of *accesses to a Web-based data-source*. An access is a lookup on a relation within a relational database, fixing values for a set of positions in the relation. For example, a transition can represent access to a Web form, where the user is restricted to filling in values for a particular set of fields. We look at verifying properties of a schema describing the possible accesses of such a system. We present a language where one can describe the properties of an access path, and also specify additional restrictions on accesses that are enforced by the schema. Our main property language, AccLTL, is based on a first-order extension of linear-time temporal logic, interpreting access paths as sequences of relational structures. We also present a lower-level automaton model, A-automata, which AccLTL specifications can compile into. We show that AccLTL and A-automata can express static analysis problems related to "querying with limited access patterns" that have been studied in the database literature in the past, such as whether an access is relevant to answering a query, and whether two queries are equivalent in the accessible data they can return. We prove decidability and complexity results for several restrictions and variants of AccLTL, and explain which properties of paths can be expressed in each restriction.

1. INTRODUCTION

Many data sources do not expose either a bulk export facility or a query-based interface, enforcing instead many restrictions on the way data is accessed. For example, access to data may only be possible through Web forms, which require bindings for particular fields in the relation [18, 4]. Querying with limited access patterns also arises in other middleware contexts (e.g. federated access to data in Web services) as well as in construction of query interfaces on top of pre-determined indexed accesses [22]. For example, a Web telephone directory might allow several Web forms that serve as *access methods* to the underlying data. It may have an access method AcM_1 accessing a relation

$$\mathrm{Mobile\#}(\underline{\mathrm{name}}, \mathrm{postcode}, \mathrm{street}, \mathrm{phoneno}),$$

where AcM_1 allows one to enter a mobile phone customer's name (the underlined field) and access the corresponding set of tuples containing a postal code, mobile phone number and street name. The same site might have an access method AcM_2 on relation

$$\mathrm{Address}(\underline{\mathrm{street}, \mathrm{postcode}}, \mathrm{name}, \mathrm{houseno})$$

allowing the user to enter a street name and postcode, returning all corresponding resident names and housenumbers. Formally an access method consists of a relation and a collection of *input positions*: for AcM_1, position 1 is the sole input position, while for AcM_2 the first two positions are input. An *access* consists of an access method plus a binding for the input positions – for example putting "Smith" into method AcM_1 is an access. The *response* to an access is a collection of tuples for the relation that agree with the binding given in the access. A schema of this sort defines a collection of *access paths*: sequences consisting of accesses and their responses.

The impact of "limited access patterns" has thus been the subject of much study in the past decade. It is known that in the presence of limited access patterns, there may be no access path that completely answers the query, and there may also be many quite distinct paths. For example, the query $\mathrm{Address}(X, Y, \text{“Jones”}, Z)$ asking for the address of Jones is not answerable using the access methods AcM_1 and AcM_2 above. There are certainly many ways to obtain the maximal answers: one could begin by obtaining all the street names and postcodes associated with Jones in the Mobile# table, entering these into the Address table to see if they match Jones, then taking all the new resident names we have discovered and repeating the process, until a fixedpoint is reached. If, however, Jones does not occur as a name in Mobile#, then this process will not yield Jones' tuple in Address. In general it is known [17] that for any conjunctive query one can construct (in linear time) a Datalog program that produces the maximal answers to a query under access patterns: the program simply tries all possible valid accesses on the database, as in the brute-force algorithm above.

In the absence of a complete plan, how can we determine which strategy for making accesses is best? Recent works [4, 3] have proposed optimizing recursive plans, using access pattern analysis to determine that *certain kinds of accesses can not extend to a useful path.* An example is the work in [3] which proposes limiting the number of accesses to be explored by determining that some accesses are not "relevant"

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 38th International Conference on Very Large Data Bases, August 27th - 31st 2012, Istanbul, Turkey.
Proceedings of the VLDB Endowment, Vol. 5, No. 3
Copyright 2011 VLDB Endowment 2150-8097/11/11... $ 10.00.

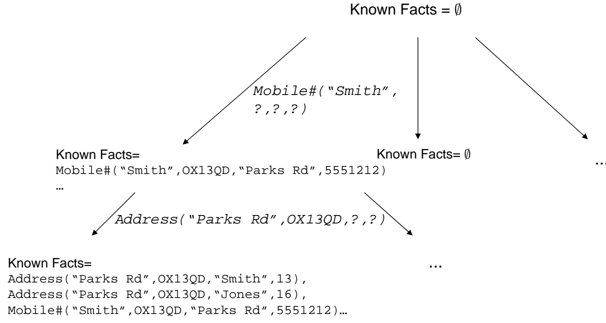

Figure 1: Tree of possible paths associated with a schema

to a query. An access is *long term relevant* if there is an access path that begins with the access and uncovers a new query result, where the removal of the access results in the new result not being discovered. [3] gives the complexity of determining relevance for a number of query languages.

Long term relevance is only one property that can be used to measure the value of making a particular access – for example we may want to know whether there is an access that reveals several values in the query result. Furthermore, "limited access patterns" represent only one possible restriction that limits the possible access paths through a web interface. Many other restrictions may be enforced, e.g:

- Restrictions that follow from *integrity constraints on the data*: e.g. a mobile phone customer name will not (arguably) overlap with a street name. Thus in an iterative process for answering the query given above, we should not bother to make accesses to the Mobile# table using street names we have acquired earlier in the process. It is also easy to see that key constraints, and more generally *functional dependencies*, can play a crucial role in determining whether an access is relevant.
- *Access order restrictions*: e.g. before making any access to Mobile#, the interface may require a web user to have made at least one access to Address.
- *Dataflow restrictions*; before making an access to Mobile# on a name, the web user must have received that name as a response to a call to Address.

Ideally, a query processor should be able to inspect an access and determine whether it is a good candidate for use, where the assumptions on the paths as well as the notion of "good candidate" could be specified on a per-application basis. In this paper we look for a general solution to specifying and determining which accesses are promising: *a language for querying the access paths that can occur in a schema.* We show that every schema can be associated with a labelled transition system (LTS), with transitions for each access and nodes for each "revealed instance" (information known after a set of accesses). A fragment of the LTS for the schema with access methods AcM_1 and AcM_2 is given in Figure 1. Paths through the LTS represent possible access/response sequences of the Web-based datasource. There are infinitely many paths – in fact every access could have many possible responses. But the access restrictions in the schema place limitations on what paths one can find in the LTS. We can then identify a "query on access paths" with a query over this transition system. This work will provide a language that allows the user to ask whether a given kind of path through instances of the schema is possible: e.g. is there a path that leads to an instance where a given conjunctive query holds, but where the path never uses access AcM_1? Is there a path that satisfies a given set of additional dataflow, access order restrictions, or data integrity constraints?

Paths are often queried with *temporal logic* [15]. We will look at natural variations of First-Order Linear Temporal Logic (FOLTL) for querying access paths. We look at a family of languages denoted $\mathrm{AccLTL}(L)$ ("Access LTL"), parameterized by a fragment L of relational calculus. It has a two-tiered structure: at the top level are temporal operators ("eventually", "until") that describe navigation between transitions in a path. The second tier looks at a particular transition, where we have first-order (i.e. relational calculus) queries that can ask whether the transitions satisfy a given property described in L. The relational vocabulary we consider for the "lower tier" will allow us to describe transitions given by accesses; it allows us to refer to the bindings of the access, the access method used, and the pre- and post-access versions of each schema relation. Consider the following AccLTL sentence:

$$(\neg\exists n\,\exists p\,\exists s\,\exists ph\ \ \mathrm{Mobile\#}_{\mathrm{pre}}(n,p,s,ph))\ \mathsf{U}$$
$$(\exists n\ \ \mathrm{IsBind}_{\mathrm{AcM}_1}(n)\wedge\exists s\,\exists p\,\exists h\ \ \mathrm{Address}_{\mathrm{pre}}(s,p,n,h))$$

The relational query prior to the "until" symbol U states that there are no entries in $\mathrm{Mobile\#}_{\mathrm{pre}}$ – the Mobile# table prior to the access. The query after the until symbol U states that an access was done with method AcM_1 and binding n, where value n appeared in the Address table prior to the access. Hence this "meta query" returns the set of access paths which have no entries revealed in relation Mobile# until an access AC is performed, where AC has method AcM_1 and uses a name that already exists in the Address table. In this work we will not be interested in returning all paths satisfying a query (there are generally infinitely many). We will check whether there is a path satisfying a given specification. This is a question of *satisfiability* for our path query language. We may also want to check that *every* path through the system is of a certain form; this is the *validity problem* for the language – bounds for validity will follow from our results on satisfiability.

We call the logic containing the above sentence $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, where $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ is the collection of positive existential queries over a signature consisting of: the access methods, bindings, and the pre- and post- access version of each relation used in a transition. $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ can express a wide variety of properties. Unfortunately we show that satisfiability for the logic is undecidable. However, we show that a rich sublanguage of $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, denoted AccLTL^{+}, has a decidable satisfiability problem. In AccLTL^{+} the formulas involving the bindings only occur positively. We give bounds on the complexity of this fragment, using a novel technique of *reduction to containment problems for Datalog.* We then look at the exact complexity of smaller language fragments, and show that the complexity can go much lower – e.g. within the polynomial hierarchy. The main thing we give up in these languages is the ability to express dataflow restrictions. We also study the complexity and expressiveness of extensions of the languages with inequalities and with branching time operators. In summary, our contributions are:

- We present the first query language for reasoning about the possible paths of accesses and responses that may appear in a Web form or other limited-access datasource.
- We show that combining a natural decidable logic for temporal data (LTL) with conjunctive queries gives an undecidable path query language.
- We show that by restricting to "binding positive" queries, we get a decidable path query language. In the process we introduce a new automaton model that corresponds to a process repeatedly querying a Web data source. We show that analysis of these "access automata" can be performed via reduction to (decidable) Datalog containment problems. The automaton and logic specification languages are powerful enough to express a rich set of data integrity constraints, access order restrictions, and data flow restrictions.
- We show that the complexity of the logic can be decreased drastically by restricting the ability to express properties of the bindings that occur in accesses. The resulting language can still express important access order and data integrity restrictions, although not dataflow restrictions.
- We determine the impact adding inequalities to the relational query language, and of adding branching operators, both in terms of expressing critical properties of accesses and on complexity of verification.

Organization: Section 2 gives the basic definitions related to access patterns, along with our family of languages $\mathrm{AccLTL}(L)$. Section 3 gives our results about the full language $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ while Section 4 deals with AccLTL^{+} and its restrictions. Section 5 discusses extensions of AccLTL^{+}. Section 6 gives conclusions and overviews related work.

2. DEFINITIONS

Schemas and paths through a schema. Let Types be some fixed set of datatypes, including at least the integers and booleans. Our schemas extend traditional relational schemas under the "unnamed perspective" [1]. A schema Sch includes a set of relations $\{S_1 \dots S_n\}$, with each S_i associated with a function from $\{1 \dots n_i\}$, where n_i is the arity of S_i, to Types. We refer to the set $\{1 \dots n_i\}$ as the *positions* of S_i and the output of the function as the *domain* of the jth position. An *instance* I for the schema consists of a finite collection $I(S_i)$ of tuples for each relation S_i, where a tuple is a function from the positions of S_i to the corresponding domain.

A schema will also have a collection of *access methods*, where each method AcM is associated with a relation S_i and a collection of *input positions* Inp(AcM). Informally, each access method allows one to input a tuple of values for Inp(AcM) and get as a result a set of matching tuples.

An *access* consists of an access method and a binding – a mapping taking the input positions of the method to their domains. A *boolean access* is one where the access method has as inputs every position of the relation – it is thus a membership test. We will use an intuitive notation for accesses, often omitting the access method. Mobile#("Jones", ?, ?, ?) is an access to relation Mobile# asking for all phone number information for people named "Jones". An example of a boolean access is Mobile#("Jones", "OX13QD", "Parks Rd", 23)?, where we add the ? to make clear it is an access.

Given an access $(\mathrm{AcM}, \bar{b})$, a *well-formed output* for AcM (on instance I) is any set of tuples r in I in the relation of AcM that is compatible with $\bar{b}$ on the input positions. We also refer to this as a *well-formed response*.

A sequence $((\mathrm{AcM}_1, \bar{b}_1), \mathrm{r}_1), \dots, ((\mathrm{AcM}_n, \bar{b}_n), \mathrm{r}_n)$ of accesses and well-formed responses for some instance I is an *access path for the instance* I. We also refer to any sequence of accesses and responses as an access path (without reference to any instance). Note that every such sequence is an access path for *some* instance – the instance containing all returned tuples. Given an access path p and an initial instance I_0 the *configuration returned by* p *on* I_0, $\mathrm{Conf}(p, I_0)$ is the instance where relation S_i contains $I_0(S_i)$ unioned with all tuples returned by any access to S_i in p. When I_0 is empty or understood from context we refer to the instance resulting from p, or $\mathrm{Conf}(p)$.

As mentioned in the introduction, one is not interested in arbitrary paths, but those satisfying additional "sanity properties". We allow our schemas to prescribe some common additional properties of access methods, while additional restrictions can be expressed in the logics. The weakest property we consider here is called *idempotence*: an access path is idempotent if whenever the path repeats the same access, it obtains the same results. This corresponds to the requirement that accesses are deterministic. A stronger property is that accesses are *exact*: an access path is exact on an instance I if for every access $(\mathrm{AcM}, \bar{b})$, the corresponding response R contains exactly the tuples in the relation of AcM which agree with $\bar{b}$ on the input positions. An access path is exact if it is exact for some input instance. Put another way, an exact access path is one that contains sound and complete views of the input data for all accesses made. Most web sources are not expected to be exact – an online music site will generally not contain information about all online music. However, some forms may be known to have canonical information – e.g. a web form accessing data from a trusted government agency. We allow situations which mix exact and non-exact accesses. In general, a schema may say that some access methods are exact, some are idempotent, and some are neither. Given a set of access methods S, we say that an access path is S-exact if there is an instance I such that the path is exact for all accesses with methods in S, and similarly talk about S-idempotence.

Finally, we often do not want paths in which values for access method inputs are "guessed", but are only interested in paths where the input to an access method is a value already known. Given an instance I_0 (representing the "initially known information") an access path $p = a_1, r_1 \dots$ is *grounded in* I_0 if every value in a binding a_i occurs either in I_0 or in a response from some a_j with $j < i$. Groundedness is a special kind of dataflow restriction – our largest logics will be able to specify groundedness, along with more specialized dataflow restrictions, but we allow them also to be imposed in the schema.

A *labelled transition system* (LTS) is of the from (No, L, T) where No is a collection of nodes, L is a collection of edge labels, and T is a collection of transitions — elements of $\mathrm{No} \times \mathrm{L} \times \mathrm{No}$. With any schema and initial instance I_0 we can associate a labelled transition system where the nodes are all the instances containing I_0 as a subinstance, the labels are all the accesses, and there is a transition $(\mathrm{I}, \mathrm{AC}, \mathrm{I}')$ whenever there is some response r to AC such that $\mathrm{Conf}((\mathrm{AC}, r), \mathrm{I}) = \mathrm{I}'$. We can also consider the restricted LTS where we only

allow paths with transitions $(\mathrm{I}, \mathrm{AC}, \mathrm{I}')$ in which the access AC is grounded at I, only paths that are idempotent, or only paths that are exact for a given subset of the access methods.

Logics for querying access paths. To query paths it is natural to use Linear Temporal Logic (LTL) [15]. LTL formulas define positions within a path. In Propositional LTL, the positions within paths are associated with a propositional model over some set of propositions, and one can then build up formulas from the propositions using the modal operators, S (since), U (until), X^{-1} (previously), X (next), and F (eventually). For example $\mathsf{F}(Q \wedge \mathsf{X}P)$ holds on positions i in a path p that come before some position j such that proposition Q holds at j and proposition P holds on position $j+1$. We want to extend LTL to deal with access paths, which are not just a sequence of propositional structures. Each position in an access path consists of an access and its response; the corresponding path through the LTS defined above consists of transitions $t_1 \ldots t_n$, where a transition t_i is of the form $(\mathrm{I}_i, (\mathrm{AcM}_i, \bar{b}_i), \mathrm{I}_{i+1})$. There is obviously a one-to-one correspondence between access paths and LTS paths as above, and we will often identify them. Since the positions carry with them a relational structure, we will use a variant of First Order Linear Temporal Logic (FOLTL) [15], which allows the use of first-order quantifiers and variables along with modal ones. We will deal here with a variant of FOLTL in which first-order sentences describing properties of positions can be nested inside temporal operators, but not vice versa.

The embedded FO formulas have the ability to constrain the instance before the access as well as afterwards. Hence, for a given vocabulary Sch, we will consider formulas over the relational vocabulary $\mathrm{Sch}_{\mathrm{Acc}}$ consisting of two copies $R_{\mathrm{pre}}, R_{\mathrm{post}}$ of each schema relation $R \in \mathrm{Sch}$. In addition $\mathrm{Sch}_{\mathrm{Acc}}$ contains predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ for each access method AcM in Sch. The arity of $\mathrm{IsBind}_{\mathrm{AcM}}$ is the number of input positions of AcM. An LTS path $t_1 \ldots t_n$ is associated with a sequence of $\mathrm{Sch}_{\mathrm{Acc}}$ structures, where the ith structure $M(t_i)$, corresponding to $t_i = (\mathrm{I}_i, (\mathrm{AcM}_i, \bar{b}_i), \mathrm{I}_{i+1})$ interprets each predicate R_{pre} using the interpretation of R in I_i, each predicate R_{post} as the interpretation of R in I_{i+1}. The predicate $\mathrm{IsBind}_{\mathrm{AcM}_i}$ holds of exactly the tuple $\bar{b}_i$ while all other predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ are empty.

We now introduce our main specification formalism, AccLTL ("Access Linear Temporal Logic").

Definition 2.1. *Let L be a subset of first-order logic over $\mathrm{Sch}_{\mathrm{Acc}}$. The logic $\mathrm{AccLTL}(L)$ has as atomic formulas every sentence of L, and is built up by the usual LTL constructors:*

$$\neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

The semantics of $\mathrm{AccLTL}(L)$ is given by the relation $(p, i) \models \varphi$, where $p = t_1 \ldots t_n$ is an LTS path and $i \leq n$. It combines the standard semantics of L formulas with the usual rules for the constructors of LTL: 1. $(p, i) \models \varphi$ iff $\varphi \in L$ and $M(t_i)$ satisfies φ in the usual sense of first-order logic. 2. $(p, i) \models \neg\varphi$ iff $(p, i) \not\models \varphi$. 3. $(p, i) \models \mathsf{X}\,\varphi$ iff $(p, i+1) \models \varphi$. 4. $(p, i) \models \varphi\,\mathsf{U}\,\psi$ iff there exists $j \geq i$ $(p, j) \models \psi$ and $\forall i \leq k < j, (p, i) \models \varphi$. 5. $(p, i) \models \varphi \vee \psi$ iff $(p, i) \models \varphi$ or $(p, i) \models \psi$.

In the rest of the paper, we make use of the temporal operators G ("globally") and F ("eventually"). These operators can be expressed using X and U as usual in LTL. The *language* of a formula φ is the set of paths p such that $(p, 1) \models \varphi$.

Our main language of interest is $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, where $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ consists of all positive existential FO sentences over the signature $\mathrm{Sch}_{\mathrm{Acc}}$.

Example 2.2 [3, 5] study query containment under (in our terminology, grounded) access patterns. Query Q_1 is contained in Q_2 relative to a schema with access patterns means that for every grounded access path p, if the configuration resulting from p satisfies Q_1, then it also satisfies Q_2. Informally, the facts about Q_1 that we can determine given the schema restrictions are contained in the facts we can determine about Q_2. Using a containment algorithm, one can perform query minimization in the presence of access restrictions.

In [5] containment under access restrictions is shown to be decidable for conjunctive queries, while [3] studies the complexity of the problem. One can see that Q_1 is contained in Q_2 under grounded access patterns iff the following $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formula is a validity (over grounded paths):

$$\mathsf{G}\,\neg\left(Q_1^{\mathrm{pre}} \wedge \neg Q_2^{\mathrm{pre}}\right)$$

Here Q_i^{pre} is obtained from Q_i by replacing each schema predicate S by S_{pre} (one could as easily use S_{post}). We will show that containment under grounded access patterns can be expressed in a restricted fragment of $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, as well as in an automaton-based specification formalism where validity relative to grounded access paths is decidable in 2EXPTIME. Our results will thus give tight bounds for containment under grounded access patterns.

Example 2.3 A boolean access AC_1 is said to be *long term relevant* [3] (LTR) for a query Q on an initial instance I_0 if there is an access path $p = \mathrm{AC}_1, \mathrm{r}_1\, \mathrm{AC}_2, \mathrm{r}_2 \ldots$ such that the configuration I resulting from applying p to I_0 satisfies Q, and the configuration resulting from the path with AC_1 dropped (i.e. $\mathrm{AC}_2, \mathrm{r}_2 \ldots$) leads to a configuration where Q does not hold. In the terminology of [3] we say it is *LTR under grounded accesses* if there is a grounded access path satisfying the above.

We claim that this property can be expressed in $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ in the following sense: for each $\mathrm{I}_0, \mathrm{AC}_1 = (\mathrm{AcM}_1, \bar{b}_1)$, and Q there is an $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formula φ which is satisfiable iff AC_1 is LTR. Below we give the formula for I_0 being the empty instance:

$$\mathsf{F}\left(\neg Q^{\mathrm{pre}} \wedge \mathrm{IsBind}_{\mathrm{AcM}_1}(\bar{b}_1) \wedge Q^{\mathrm{post}}\right)$$

The formula checks that there is a path p and a response r_1 to AC_1, such that Q holds after p but not after $p, \mathrm{AC}_1, \mathrm{r}_1$. But for a boolean access AC_1, the instance after $p, \mathrm{AC}_1, \mathrm{r}_1$ is the same as the one after $\mathrm{AC}_1, \mathrm{r}_1, p$.

As mentioned in the introduction, we often want additional data integrity restrictions on the path. In $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, we can add on many data integrity restrictions, such as the disjointness of names from streets, which would be expressed by a conjunction of several formulas, including:

$$\mathsf{G}(\neg\exists n\,\exists p\,\exists s\,\exists ph\,\exists hn\,\exists n'\,\exists pc\ \ \mathrm{Mobile\#}_{\mathrm{pre}}(n, p, s, ph) \wedge\ \mathrm{Address}_{\mathrm{pre}}(n, pc, n', hn))$$

Similarly we can add on access order restrictions and dataflow restrictions. For example, the following would restrict to

paths in which names input to Mobile# must have appeared previously in Address:

$$\mathsf{G}((\exists n\ \mathrm{IsBind}_{\mathrm{AcM}_1}(n)) \rightarrow \exists n\,\exists s\,\exists hn\,\exists pc\ \mathrm{IsBind}_{\mathrm{AcM}_1}(n) \wedge \mathrm{Address}_{\mathrm{pre}}(s, pc, n, hn))$$

Example 2.4 (Data integrity restrictions, continued) Let Sch be a schema that includes, in addition to the access methods, a collection of functional dependencies $d_i = R^i : pos_i \rightarrow a_i$, where pos_i are positions of R^i and a_i is a position of R^i. We say that an access AcM is Long-term relevant for Q under Sch if there is an instance $\mathrm{I} \supseteq \mathrm{I}_0$ satisfying all the FDs and an access path that reveals Q to be true, as in Example 2.3, but where each response returns only tuples in I.

This can be expressed in AccLTL($L_\exists^{\neq}$), where $L_\exists^{\neq}$ is the set of conjunctive queries with inequalities.

$$\mathsf{F}\left(\neg Q^{\mathrm{pre}} \wedge \mathrm{IsBind}_{\mathrm{AcM}}(\bar{b}_1) \wedge Q^{\mathrm{post}}\right) \wedge \bigwedge_i \neg \mathsf{F}[\exists \vec{y}\, \vec{y}'\ R^i_{\mathrm{pre}}(\vec{y}) \wedge R^i_{\mathrm{pre}}(\vec{y}') \wedge \bigwedge_{k\,\in\,\mathrm{pos}_i} y_k = y'_k \wedge y_{a_i} \neq y'_{a_i}]$$

where Q^{pre} and Q^{post} are defined as in the previous example. We will look at languages with inequalities in Section 5.

Basic Computational Problems. The basic problem we consider is satisfiability of a sentence φ, which by default means that there is some access path p such that $(p, 1) \models \varphi$. We will also consider satisfiability over grounded, idempotent, and (S-) exact paths.

3. AN EXPRESSIVE LANGUAGE FOR ACCESS RESTRICTIONS

Since satisfiability for first-order logic is undecidable, it is clear that AccLTL(FO) has an undecidable satisfiability problem. Our first main result is that the same holds even when first-order formulas are restricted to be existential.

THEOREM 3.1. *The satisfiability problem for* AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) *is undecidable*

This is surprising, in that AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) formulas deal with a fixed set of existential sentences on the configuration, and as a path progresses these queries can only move from false to true as more tuples are exposed by accesses.

The proof works by reducing the problem of determining whether a collection Γ of functional dependencies (fds) and inclusion dependencies (ids) implies another functional dependency σ. Since this problem is known to be undecidable [6], it suffices to reduce it to unsatisfiability of a AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) formula.

The difficulty here is that functional dependencies seem to require negation inside a universal quantification, while inclusion dependencies require quantifier alternation – in AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) we have only boolean combinations of positive formula. We now explain the main idea involved in bridging this gap, which will also be used in later undecidability arguments (Theorem 5.2). The schema for our accesses includes a successor relation of a total order over the tuples of each relation in $\Gamma \cup \{\sigma\}$. The successor relation is "created" via accesses – that is, we perform accesses that reveal associations between a tuple and its successor. For each relation R mentioned in $\Gamma \cup \{\sigma\}$ we also have relations Beg(R) and End(R). Our formula will enforce that that these contain the first and the last tuples in the total order, respectively, by asserting the existence of additional accesses to these relations that reveal the first and last tuple. After all the relations are filled, the satisfaction of the different fd's and id's in Γ and the failure of σ are verified. The satisfaction of the dependencies makes use of the successor relation, and we explain the idea for FDs. We verify a dependency for one tuple at a time, iterating on the tuples according to the order. We will use a new predicate $\mathrm{Chk}^{\mathrm{FD}}(R)$ whose arity is twice the arity of R. This predicate will have a boolean access. $\mathrm{Chk}^{\mathrm{FD}}(R)(\vec{t}, \vec{t}')$ holding at some instance indicates that $\vec{t}, \vec{t}'$ is in accordance with the FDs on R. This will be done in a "nested loop" (a pair of nested "untils" in the logic) in which we iterate first over tuples $\vec{t}$, then over tuples $\vec{t}'$, accessing them progressively within $\mathrm{Chk}^{\mathrm{FD}}(R)$. At every access, we check whether the FD is satisfied, and if it is we continue the iteration. Details are in the appendix.

4. VERIFIABLE SPECIFICATIONS: THE POSITIVE TRANSITION SUBLANGUAGE

The undecidability proof of AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) makes use of the ability of the logic to enforce that an access is made to a binding that does *not* satisfy a certain relation. We now consider a restriction of AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) which adds an additional monotonicity condition. A formula φ in AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$) is *binding-positive* if every atom of the form IsBind($\vec{w}$) occurs only positively in φ – that is, under an even number of negations.

DEFINITION 4.1. *The logic* AccLTL^+ *is the set of binding-positive formulas in* AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$).

Note that in AccLTL^+ we can describe the most basic dataflow constraint, the property of an access path being grounded: an access is grounded iff for every transition in a path, for every value that occurs in a binding, it occurs in some relation in the instance prior to the access:

$$\mathsf{G}\left(\exists \vec{x}\ \ \mathrm{IsBind}_{\mathrm{AcM}}(x_1 \ldots x_m) \wedge \bigwedge_{i\,\leq\,m} \bigvee_{R\,\in\,\mathrm{Sch}} \exists \vec{y}\, R(y_1 \ldots y_n) \wedge \bigvee_{j\,\leq\,n} y_j = x_i\right)$$

Thus we can reduce satisfiability over grounded instances to satisfiability over all instances. Furthermore all the examples in the introduction are expressible in this fragment; we can express relevance of an access to a query as well as containment of queries under access patterns, restricting the paths to satisfy many data integrity, dataflow, and access ordering restrictions.

Our next main result is that this restriction suffices to give decidability:

THEOREM 4.2. *Satisfiability of* AccLTL^+ *is decidable in 3EXPTIME. The same is true for satisfiability over grounded instances and satisfiability over idempotent and exact accesses.*

We will show Theorem 4.2 by going through another specification formalism of interest in its own right, a natural automaton model for access paths. These are *Access-automata*

(A-automata for short), which run over access paths, using a finite set of control states. At each transition $(\mathrm{I}, (\mathrm{AcM}, \bar{b}), \mathrm{I}')$ of an access path the evolution function of the automaton tells what new states (if any) it can move to at the next position. The evolution function is a relational query that makes use of the binding, pre- and post- condition of the transition.

Definition 4.3 (A-Automaton). *Let* Sch *be a schema,* $\mathrm{Sch}_{\mathrm{Acc}}$ *the corresponding schema with accesses (as defined in Section 2), and C a set of constants. An* Access-automaton *(A-automaton for short) over* (Sch, C) *is a tuple* (S, s_0, F, δ) *where*

- *S is a finite set of states, $s_0 \in S$ is an initial state, $F \subseteq S$ is a set of accepting states*
- *δ is a finite set of tuples of the form $(s, \psi^- \wedge \psi^+, s')$ where s, s' are states, ψ^- is a positive boolean combination of negated $\mathrm{FO}^{\exists +}_{\mathrm{Acc}}$ sentences that can not mention the predicate* IsBind*, while ψ^+ is a $\mathrm{FO}^{\exists +}_{\mathrm{Acc}}$ sentence; all these formulas can use constants in the given set C.*

Semantics. Let $A = (S, s_0, F, \delta)$ be an A-automaton and let p be a path $t_1 \dots t_n$ through the LTS associated with Sch, where $t_i = (\mathrm{I}_i, (\mathrm{AcM}_i, \bar{b}_i), \mathrm{I}_{i+1})$. A *run* of A on p assigns to every t_i a δ_i of the form $(s_i, \varphi_i, s_{i+1})$ in δ so that the relational structure $M(t_i)$ associated with t_i satisfies φ_i. A run of A is further said to be accepting iff its first state is initial and its last state is final. The language $L(A)$ accepted by an A-automaton A is the set of access paths for which there is an accepting run. Note that an automaton only accepts access paths, which by definition must satisfy at least the property that for each i, I_{i+1} extends I_i solely by adding tuples to the relation of AcM_i, and all tuples added are consistent with the binding on the input positions of AcM_i. The definition of $L(A)$ can be further qualified to account for other sanity conditions (e.g. exactness).

A-automata are powerful enough to directly express relevance of an access in the presence of dataflow restrictions as well as disjointness constraints. In particular, the following is easy to show (and is proven in the appendix):

Proposition 4.4. *Let Q and Q' be two positive queries, ACS a set of access methods, and Σ a set of disjointness constraints. One can efficiently produce an A-automaton A such that Q is contained in Q' under limited access patterns with disjointness constraints iff the language recognized by A is empty. A similar statement holds for long-term relevance of an access to Q under disjointness constraints.*

The proposition above can be extended to a general result stating that high-level logical specifications can be compiled into A-automata. We say that an A-automaton A is equivalent to an AccLTL sentence φ if the language of the φ is the same as the language of A. The following result shows that each AccLTL^+ formula can be converted into an A-automaton.

Lemma 4.5. *For each* AccLTL^+ *formula φ there is an equivalent A-automaton of size exponential in the size of φ.*

We will show that emptiness of A-automata is decidable. Note that this decidability result together with Lemma 4.5 completes the proof of Theorem 4.2. Again, there are variants of the theorem for the various types of access, but we focus on the case of general accesses in the body of the paper.

Theorem 4.6. *Emptiness of A-automata is decidable in 2EXPTIME. The same holds if accesses are restricted to be exact or idempotent.*

Notice that from Theorem 4.6 and Proposition 4.4 we get a 2EXPTIME upper bound for containment and long-term relevance. This improves on the prior known bounds [3, 5].

The proof uses a tight connection between A-automata and the containment problem for Datalog queries within positive first-order queries. This connection can also be exploited to give a corresponding lower bound:

Theorem 4.7. *Emptiness of A-automata and satisfiability of* AccLTL^+ *are both 2EXPTIME-hard.*

4.1 Automata, Datalog, and proof sketch of Theorem 4.7

The proof of this result makes use of some new tools that we overview here. We reduce the emptiness problem for A-automata to the problem of whether a Datalog program is contained within a positive first-order query. Roughly speaking, we show that these automata can be captured by a conjunction of a Datalog query and the negation of a union of conjunctive queries. The reduction to this problem involves several stages, and the first step goes through a syntactic subclass of A-automata, called "progressive A-automata", defined below. We will show that the problem of testing emptiness of A-automata can be reduced to checking the emptiness of a bounded number of progressive A-automata.

Progressive Automata. In the following, given a boolean combination of $\mathrm{FO}^{\exists +}_{\mathrm{Acc}}$ formulas φ, we denote by $\tilde{\varphi}$ the formula $\exists \bar{x}\ \varphi'$ where φ' is obtained from φ by replacing each atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ by $\bar{t} = \bar{x}$ and by replacing each predicate R_{pre} by R_{post}. For a set Φ of sentences, we say that a formula is a *complete Φ-type* if it is a conjunction that contains every formula of Φ either positively or negated. A formula is a "pure pre" (resp. "pure post") formula if it only mentions predicates of the form R_{pre} (resp. R_{post}).

Definition 4.8 (Progressive A-automaton). *An A-automaton $A = (S, s_0, F, \delta)$ over* (Sch, C) *is* progressive *if there is a pure pre formula $\Upsilon_{\mathrm{pre}}(s_0)$ that does not use the predicate* $\mathrm{IsBind}_{\mathrm{AcM}}$*, a set of pure post $\mathrm{FO}^{\exists +}_{\mathrm{Acc}}$ sentences Φ, and a function Υ_{post} mapping the states of A to complete Φ-types such that:*

1. *For any transition (s, φ, s'), if both $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ and $\mathrm{IsBind}_{\mathrm{AcM}'}(\bar{t}')$ are atoms in φ, then $\mathrm{AcM} = \mathrm{AcM}'$.*
2. *For any transition (s, φ, s'), φ implies $\Upsilon_{\mathrm{post}}(s')$.*
3. *For any transition (s_0, φ, s') that leaves the initial state, φ implies $\Upsilon_{\mathrm{pre}}(s_0)$*
4. *For any transition (s, φ, s') for which s and s' are in the same strongly connected component, $\Upsilon_{\mathrm{post}}(s)$ is equivalent to $\Upsilon_{\mathrm{post}}(s')$; also $\Upsilon_{\mathrm{post}}(s')$ implies $\tilde{\varphi}$.*
5. *The maximal strongly connected components of A form a sequence $C_1, \dots, C_h$. That is, for each $i < h$, there is exactly one transition (s, φ, s') such that $s \in C_i$ and $s' \in C_{i+1}$. For such a transition that connects two maximal strongly connected components, all atoms of the form $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ must not contain variables; that is, $\bar{t}$ must be a sequence of constants.*

6. *The initial state is in C_1 and all accepting states are in C_h.*

We will call h the *height* of A. The following lemma, proven in the appendix, shows that A-automata correspond, up to emptiness, to unions of progressive automata.

LEMMA 4.9. *For every A-automaton A, there are progressive A-automata $A_1, \ldots, A_n$, such that, for each $i \leq n$, the size of A_i is polynomial in the size of A, n is exponential in the size of A, and $L(A)$ is empty iff $L(A_1) \cup \ldots \cup L(A_n)$ is empty.*

From progressive A-automata to containment of Datalog in Positive Queries. We now proceed to show that emptiness of progressive A-automata is decidable. Together with Lemma 4.9 this implies the decidability of (general) A-automata. This will involve reducing the emptiness of a progressive A-automaton to the problem of whether a Datalog program is contained in a positive first order logic sentence.

Recall that a Datalog program is defined with respect to two database schemas, called the *extensional schema* and the *intensional schema*. A *Datalog program* $\mathcal{P}$ is a finite set of rules of the form "head $:-$ body" where head is an atomic formula $R(\bar{x})$ with a relation symbol R in the intensional schema, and where body is a conjunctive query that can use relation symbols from the intensional and the extensional schema. Each Datalog program $\mathcal{P}$ contains a distinguished *goal predicate* Q. We use the standard notions of the least fixedpoint of a Datalog program $\mathcal{P}$ on a database D (see [1]), and we denote this fixedpoint by $\mathcal{P}(D)$. We say that a Datalog program $\mathcal{P}$ *accepts* a database D if the goal predicate of $\mathcal{P}$ is not empty in $\mathcal{P}(D)$.

LEMMA 4.10. *Let A be a progressive A-automaton. Then there exists a Datalog program $\mathcal{P}_A$ and a positive first order logic sentence $\mathcal{P}'_A$ such that $L(A)$ is not empty iff $\mathcal{P}_A$ is not contained in $\mathcal{P}'_A$. One can construct these in polynomial time in the size of A.*

The proof of this lemma is itself quite involved. The basic idea of this proof is that $\mathcal{P}_A$ enforces the positive constraints of A while $\mathcal{P}'_A$ enforces the negative constraints. Recall that in a progressive automaton, the evolution is in a fixed number of stages, based on the number of subqueries satisfied. A stage represents a strongly connected component of the automaton. The extensional database D will have predicates $\mathrm{BackgroundR}_i$ representing the part of relation R that becomes visible to A at the end of each stage i, along with predicates $\mathrm{IntBackgroundR}_i$ representing the data that becomes visible when crossing from one stage to the next. The important intensional predicates ViewR_i will represent intermediate stages of the predicates $\mathrm{BackgroundR}_i$ within the evolution of each stage. The Datalog program $\mathcal{P}_A$ will have rules corresponding to the evolution of ViewR_i by adding tuples from $\mathrm{BackgroundR}_i$. To ensure that the tuples correspond to some valid binding, $\mathcal{P}_A$ will have rules guaranteeing that only tuples that satisfy the appropriate formulas can be added to ViewR_i. We can do this with a Datalog program by adding appropriate intermediate relations, exploiting the fact that the constraints on the guards are positive, and hence represented in non-recursive Datalog.

The role of the positive query $\mathcal{P}'_A$ is twofold: First, $\mathcal{P}'_A$ will enforce the negated conjunctive queries in the transitions – in particular, $\mathcal{P}'_A$ will contain constraints on the relations $\mathrm{BackgroundR}_i$ and $\mathrm{IntBackgroundR}_i$ that enforce that these only contain tuples that satisfy these negated constraints. In this way, whenever the Datalog program adds tuples to the intensional relations, these tuples are guaranteed to satisfy the corresponding negative constraints. The second purpose of $\mathcal{P}'_A$ is to enforce that for each i, only one relation among the $\mathrm{IntBackgroundR}_i$ is non-empty. This is important, as these relations contain the tuples that the Datalog program might add when simulating the automaton transitioning from one strongly connected component to the next. On such a transition an A-automaton can only perform one access, and hence the Datalog program should only be able to add tuples from one relation $\mathrm{IntBackgroundR}_i$ into ViewR_i.

In the proof that our construction is correct, we show that our Datalog program $\mathcal{P}_A$ can be decomposed into subprograms $\mathcal{P}_1, \ldots, \mathcal{P}_h$ that correspond to the decomposition of the A-automaton into strongly connected components $C_1, \ldots, C_h$, in the following sense: Whenever an A-automaton has a run that ends in its strongly connected component C_i, $i \leq h$ then the subprogram $\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_i$ of $\mathcal{P}$ adds tuples to the intensional database that correspond in a certain way to the tuples that A has obtained using accesses.

The details of the construction, and a proof of its correctness, are in the appendix.

Completion of the proof of Theorem 4.6. Let us review what we have accomplished thus far: we have reduced questions about our logic to non-emptiness of the automata, and non-emptiness of an automaton we have reduced to determining whether a Datalog program is contained in a positive query. To complete the proof of Theorem 4.6 we need the following new result, that generalizes a theorem of Chaudhuri and Vardi [7]:

PROPOSITION 4.11. *The containment problem of a Datalog program P in a positive first-order sentence φ, where both P and φ may make use of constants, is in 2EXPTIME.*

The proof of this result is in the appendix. Theorem 4.6 follows from the proposition and the reduction given earlier.

4.2 Restricted binding predicates and reduction to propositional LTL

We now look for path query languages where the satisfiability problem has lower complexity. We will do this by giving up the ability to talk about the exact dataflow from data instances to bindings. This will allow us to get verification algorithms based on reduction to standard Propositional Linear Temporal Logic verification, a well-studied problem for which many tools are available [9].

For a relational schema Sch, we define the vocabulary $\mathrm{Sch}_{0-\mathrm{Acc}}$ as in $\mathrm{Sch}_{\mathrm{Acc}}$ but instead of n-ary predicates $\mathrm{IsBind}_{\mathrm{AcM}}$, we have only a 0-ary predicate $\mathrm{IsBind}_{\mathrm{AcM}}$. A transition $t_i = (\mathrm{I}_i, (\mathrm{AcM}_i, \bar{b}), \mathrm{I}_{i+1})$ is now associated with the relational structure $M'(t_i)$ in which S_{pre}, S_{post} are interpreted as before, and $\mathrm{IsBind}_{\mathrm{AcM}}()$ holds exactly if $\mathrm{AcM} = \mathrm{AcM}_i$. We will now consider $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$, in which the first-order formulas use only $\mathrm{Sch}_{0-\mathrm{Acc}}$. That is, in the logic we can refer to which access was performed, but can not express anything about the bindings used.

Going back to Example 2.2 and 2.3 we say that the basic relevance properties are in this language, *provided* that we do not impose any dataflow restrictions – including any restrictions that access paths are grounded. On the other

hand, we can still impose the access order restrictions of Example 2.3. We now see that by curtailing the expressiveness, the complexity goes down significantly.

THEOREM 4.12. *Satisfiability of an* AccLTL($\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$) *formula (over all access paths) is* PSPACE*-complete. The same holds if particular access methods must be exact or idempotent.*

PROOF. The PSPACE-hardness of our problem comes from the PSPACE-hardness of the satisfiability problem of a LTL formula over finite words [15]. The upper bound is proven by bounding the size of the underlying data, and then applying results about propositional LTL.

We now prove the upper bound, focusing on the case of general access paths. Let Sch be a schema, and φ be a formula of AccLTL($\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$). First, we demonstrate that if there exists an access path that satisfies φ then there exists one where the size of each instance is bounded by a polynomial function in the sizes of φ and Sch.

The key is the following "Boundedness Lemma":

LEMMA 4.13. *An* AccLTL($\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$) *formula φ is satisfiable iff there exists a path ρ which satisfies the following properties: 1. The instances in ρ have sizes bounded by a polynomial function in the sizes of φ, and* Sch*. 2. The set of bindings used in ρ has size bounded by a polynomial function in the sizes of φ*

PROOF. Let some φ be given. Suppose that φ is satisfiable. Then there exists a path ρ that satisfies φ. We define the *positive sentences* of φ to be the maximal subsentences of φ that belong to $\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$. Consider the following rewrite rules: for each AcM $\in$ Sch we replace the formula $\mathrm{IsBind}_{\mathrm{AcM}} \wedge \psi$, where $\mathrm{IsBind}_{\mathrm{AcM}}$ is a predicate, by the formula ψ. We also replace the formula $\mathrm{IsBind}_{\mathrm{AcM}} \vee \psi$ where $\mathrm{IsBind}_{\mathrm{AcM}}$ is a predicate by the formula ψ. We denote by $Q_f(\varphi)$ the set of $\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$ sentences that have been obtained from a positive sentence of φ by inductively applying the above rules until there are no more occurrences of predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ in the result.

Let $\{q_1, \ldots, q_m\}$ be the set of sentences appearing in $Q_f(\varphi)$ that are satisfied by the last instance I_n. Let $\rho_{i_1}, \ldots, \rho_{i_m}$ be the set of transitions in the path ρ such that ρ_{i_j} is the minimal transition in ρ that satisfies q_j. Let h_j be a homomorphism from q_j to ρ_{i_j}. We let $(\mathrm{I}_{f-1}, \mathrm{AC}_f, \mathrm{I}_f)$ be the last transition in ρ. Let I'_f be the minimal subinstance of I_f such that for all i $h_i(q_i) \subseteq (\mathrm{I}'_f)^{pre} \cup (\mathrm{I}'_f)^{post})$, where for any instance I of the original schema, I^{pre} is obtained from I by interpreting relations R_{pre} by the interpretation of R in I, while $\mathrm{I}^{\mathrm{post}}$ is obtained from I by interpreting relations R_{post} by the interpretation of R in I.

Since we only need to consider witnesses to positive queries, it is easy to check that I'_f can be constructed and has size polynomial in the sizes of φ and Sch. We can thus construct a path ρ' that contains the intersection of the instances of ρ with the instance I'. ρ' satisfies φ, and the size of the instances of ρ' are bounded by a polynomial function in the size of φ and Sch.

We now restrict the bindings used in ρ'. Let p be a path. An access $(\mathrm{AcM}_i, \bar{b}_i)$ is *necessary* for p if new tuples are returned by it (i.e. tuples not in the previous instance within p), and *unnecessary* otherwise. Note that if we have a path and we change the binding on some unnecessary access to anything of the appropriate arity, while returning emptyset, then it is still a valid access path.

So without loss of generality, we can arrange that the set of bindings used in ρ' consists of the necessary accesses in ρ' plus a single binding for each access method, used in place of every unnecessary access on that method. Therefore the set of bindings is a set of tuples having size bounded by a polynomial function in the sizes of φ and the schema. □

Given the lemma, we can now apply the following NPSPACE algorithm:

1. First, we guess a finite sequence of instances $I_1 \ldots I_n$ and a sequence of accesses A, each of polynomial size (with the polynomial given by Lemma 4.13). In the remaining steps, we will check whether there is a witness path using the bindings of these accesses and only these instances.
2. We translate the AccLTL($\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$) formula φ into an ordinary LTL formula $\overline{\varphi}$ in a propositional alphabet that encodes information about which of the instances and bindings are used. This formula will be constructed so that it is satisfiable over words iff φ is satisfiable.
3. Then, we apply any PSPACE algorithm for LTL satisfiability of $\overline{\varphi}$ over finite words.

We now explain in more detail the translation to ordinary LTL that is the key step in the high-level algorithm above. Fix a sequence $s = \mathrm{I}_1 \ldots \mathrm{I}_n$ of distinct instances as well as a sequence of accesses A, both of polynomial size. We denote by B, the union of the set of bindings used in A and the set $\cup_{\mathrm{AcM}}\{b_{\mathrm{AcM}}\}$ where b_{AcM} is a binding of AcM using some values appearing in B.

We associate propositions with transitions of any of the following forms:

- Transitions of form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$ where $\vec{b}$ is in B and compatible with AcM.
- Transitions of form (I_i, A_i, I_{i+1})

The set of transitions of the above forms is denoted $T(I, B)$. For each i, we denote by $T(i)$ the set of transitions of the form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$. For each i, we denote by $t_{i,\rightarrow}$ the transition $(I_i, A(i), I_{i+1})$. For each i, we denote by $P(i)$ the set of propositions associated with the transitions of the form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$. For each i, we denote by $p_{i,\rightarrow}$ the proposition associated with the transition (I_i, A_i, I_{i+1}). The set of all such propositions is denoted Σ. The words described by $\overline{\varphi}$ are over alphabet 2^{Σ}. Intuitively, each letter of a word would be used to describe a transition $(\mathrm{I}, (\mathrm{AcM}, \bar{b}), \mathrm{I}')$.

We now describe the construction of $\overline{\varphi}$.

First, we describe some "sanity axioms" stating that a run associated with $\overline{\varphi}$ really corresponds to some access path. This requires:

- Every position has exactly one proposition of Σ.
- The order of the instances in s is respected. This is expressed by the formula:

$$\bigwedge_{i,p \in P(i)} \mathsf{G}\left(p \Rightarrow (\bigvee_{p' \in P(i)} p' \,\mathsf{U}\, p_{i,\rightarrow})\right) \wedge$$

$$\bigwedge_{i} \left(p_{i,\rightarrow} \Rightarrow \mathsf{X}(\bigvee_{p'' \in P(i+1)} p'' \vee p_{i+1,\rightarrow})\right) \wedge$$

$$(\bigvee_{p''' \in P(0)} p''' \vee p_{0,\rightarrow})$$

Next we rewrite φ to $\overline{\varphi}$ by replacing each positive sentence q of φ by the union over of $p \in \Sigma$ over all the previous transitions that satisfy it.

We claim that the $\overline{\varphi}$ is satisfiable over ordinary words iff φ is satisfiable over access paths that conform to the sequence s and the bindings in B. The direction from right to left requires taking an access path and performing the obvious propositional abstraction. In the other direction, we take a propositional word $w_1 \dots w_n$ satisfying $\overline{\varphi}$. The first sanity axiom implies that exactly one transition proposition p is associated with w_i. The second sanity axiom implies that the instance reached in the transition associated with $w(i)$ is the same as the initial instance of the transition associated with $w(i+1)$. One can check that this gives the required access path for φ.

Restricting LTL operators. Let LTL_X be the subset of LTL that only uses the temporal operator X. We denote by $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ the corresponding sublanguage of $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$.

$\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ is extremely limited in expressiveness, since it can only talk about paths of some fixed length. However, there are properties for which such small paths are sufficient. Consider Example 2.3. It is easy to see that Q is LTR over all accesses iff it is LTR over access paths of size $|Q|$ – a counterexample to long-term relevance has only polynomially length. But LTR over small paths can be expressed in $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$. Thus $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ is sufficient to tell whether an access might have an impact on answering a query, but without taking into account of even the most basic dataflow restriction on paths.

THEOREM 4.14. *Satisfiability of* $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ *is* Σ^P_2*-complete, even when certain accesses are restricted to be exact or idempotent.*

Hardness. The non-containment of positive relational queries, where positions can be restricted to have finite (i.e. enum) datatypes can be reduced to the unsatisfiability problem of either language – this problem is known to be Π^P_2-hard.

Upper-Bound. Let an $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ formula φ be given. We first note that Lemma 4.13 also holds for the logic $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$. Using this we can reduce to the language propositional LTL_X, which has a satisfiability problem in NP. In the reduction we will again guess a small number of small instances and bindings, and we will also guess which positive queries of φ will be true – this guess will then be verified via a sequence of NP (for queries guessed to be true) and co-NP (for queries guessed to be false) subroutines. We can then rewrite the original formula φ to an LTL_X formula that is satisfiable iff φ is satisfiable on a sequence based on the guessed instances and bindings. Details are in the appendix.

5. EXTENSIONS AND LIMITS

We look at the impact of two natural extensions on our decidability results: allowing inequalities and branching formulas.

5.1 Extension to Inequalities

Our results on decidable fragments did not use inequalities, and inequalities are useful for expressing data integrity constraints. The most obvious example involves keys and functional dependencies, as discussed in Example 2.4.

By making a straightforward modification of the proofs without inequalities, we can see that inequalities add nothing to the complexity of $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ and its sublanguages.

THEOREM 5.1. *Letting* $\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}}$ *be the language of positive queries with inequalities over the restricted vocabulary with only the* 0*-ary predicates* $\mathrm{IsBind}_{\mathrm{AcM}}$*, we have that*

- *satisfiability of* $\mathrm{AccLTL}(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$ *is in* PSPACE *(and hence* PSPACE*-complete by Theorem 4.12)*
- *satisfiability of* $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$ *is in* Σ^P_2 *(hence* Σ^P_2*-complete by 4.14)*

Using the language above, one can express relevance or containment in the presence of functional dependencies, access order constraints, and disjointness constraints, but not dataflow constraints.

For the language AccLTL^+, shown decidable in Theorem 4.2, inequalities make a dramatic difference. The proof of the theorem below shows that we cannot capture both dataflow restrictions like groundedness along with rich integrity constraints such as functional dependencies, while retaining decidability. The proof also shows that many extensions of AccLTL^+ with aggregation – basically, any that are expressive enough to capture FDs – will be undecidable.

THEOREM 5.2. *For binding-positive* $\mathrm{AccLTL}(\mathrm{FO}^{\exists+,\neq}_{\mathrm{Acc}})$*, satisfiability is undecidable*

PROOF. Again we reduce the problem of implication of functional dependencies (fds) and inclusion dependencies (ids) for relational databases to the problem of the unsatisfiability of a AccLTL^+ extended with inequalities.

Let Γ be a set of inclusion and functional dependencies, and σ be a functional dependency over Sch.

The approach to the reduction is similar to that in Theorem 3.1. We will make iterative accesses to a successor relation of a total order over the tuples. We will also access relations $\mathrm{Beg}(R)$ and $\mathrm{End}(R)$, and verify that they contain the first and the last tuples of relation R according to the order. While iterating through the relations according to the successor relation, the satisfaction of the different fd's and id's and the failure of σ are verified. The satisfaction and failure of fd's can be reduced to the satisfaction of a boolean combination of conjunctive queries with inequalities – the successor relation is not needed. The satisfaction of an inclusion dependency id whose source is a relation R is where we use the successor relation, and the iteration technique of Theorem 3.1. Again, it is easy to check an inclusion dependency for a source relation consisting of only a single tuple, since this requires only existential quantification. We verify an id on source relation R by checking for witnesses for one tuple in the source of the dependency at a time, iterating on the tuples according to the successor relation. We will use a new predicate $\mathrm{CheckIncDep}(id)$ whose arity is the arity of R. $\mathrm{CheckIncDep}(id)(\vec{t})$ holding at some instance indicates that $\vec{t}$ has been verified to satisfy the inclusion dependency id. This will be done in a "loop" (an "until" in the logic) in which we look for a tuple $\vec{t}$ whose predecessor in the order satisfies $\mathrm{CheckIncDep}(id)$, and which satisfies the inclusion dependency; when we find such a tuple, we perform an access to $\mathrm{CheckIncDep}(id)$ on it. At the end of this

"loop", we check that the final tuple in the ordering satisfies CheckIncDep(id). Details are in the appendix. □

The reader may want to look at Figure 2 for a view to how the languages with inequalities relate to the languages defined previously.

5.2 Branching time formulas

Thus far we have discussed only linear time properties of the LTS of a schema with access relations. What about branching time logics, which can consider the relationship of multiple paths? For example, a branching time logic could express that we have reached a point where no further information about boolean query Q can be obtained without guessing values to enter into forms – e.g. there are possible worlds consistent with the known facts where Q is true and also consistent worlds where Q is false, but the truth of Q can not be revealed by any further sequence of grounded accesses. Unfortunately, we will show that even very limited branching time expressiveness leads to undecidability.

Let L be a fragment of first-order logic over the smallest vocabulary we have considered thus far: two copies $S_{\mathrm{pre}}, S_{\mathrm{post}}$ of each relation symbol S and the proposition $\mathrm{IsBind}_{\mathrm{AcM}}$.

We will consider a small fragment of branching time logic built up from L-formulas, analogously to the way we built up AccLTL formulas over sentences of L in the linear time logic. Traditional branching time logic allows the combination of path quantification with modal operators. In our setting we will consider a very simple kind of branching, which looks ahead only one step – we will refer to it as $\mathrm{CTL}_{EX}(L)$, but instead of CTL we might as easily have said "basic modal logic" or Hennessy-Milner Logic [15], since we only need the power of the most basic existential modality to get undecidability. $\mathrm{CTL}_{EX}(L)$ has the rules: every L sentence is a formula, boolean combinations of formulas are formulas, and if φ is a formula then $\mathsf{EX}\varphi$ (in modal logic notation, $\diamond\varphi$) is a formula.

The semantics is defined as a relation $(S,t) \models \varphi$, where t is a transition $(\mathrm{I}, \mathrm{AC}, \mathrm{I}')$ in the labelled transition system S associated with a schema Sch. When φ is an L formula, this holds iff the relational structure associated to t, $M'(t)$, satisfies φ in the usual sense of first-order logic. The semantics of boolean operators is the usual one. Finally, $(S,t) \models \mathsf{EX}\varphi$ iff there is a successor t' of t such that $(S,t') \models \varphi$. Note that instead of referring to CTL here, we could have used basic modal logic or Hennessy-Milner Logic. Note that Deutsch et. al. [14] have shown undecidability for some branching time logics over LTS's associated with a similar model of relational transducers – but in their case the logics (e.g. Theorem 4.14 of [14]) allow one to describe properties of the input (analogous to our larger signature $\mathrm{Sch}_{\mathrm{Acc}}$), while here we can only describe the access propositionally.

We show that even this restricted logic is undecidable, even when the base formulas are existential.

THEOREM 5.3. *Satisfiability of* $\mathrm{CTL}_{EX}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ *formulas is undecidable*

PROOF. We reduce from the problem of implication of a functional dependency (FD) from a set of functional dependencies and inclusion dependencies (IDs) for relational databases. This is known to be undecidable [6].

Let Γ be a set of inclusion and functional dependencies over a relational schema Sch and σ an FD. For simplicity, we will assume all positions in the schema have the same type (say, integer type). We will first extend Sch with additional relations, along with access patterns.

For each relation R of Sch, we have an access method Fill_R on R with no inputs. Thus each access $(\mathrm{Fill}_R, \emptyset)$ returns an essentially random configuration of R. We also have additional relations $\mathrm{Chk}^{\mathrm{FD}}(R)$, having twice the arity of R and CheckIncDep(R) having the same arity as R. We have boolean access methods on all of these additional relations – that is, methods where all positions are in the input.

Our reduction will create a formula $\psi(\Gamma,\sigma)$ of the form:

$$\mathsf{EX}\Big(\mathrm{Fill}_{R_1} \wedge \mathsf{EX}\big(\cdots \wedge \mathsf{EX}(\mathrm{Fill}_{R_n} \bigwedge_{\mathrm{fd}\in\Gamma} \varphi_{\mathrm{fd}} \wedge \bigwedge_{\mathrm{id}\in\Gamma} \varphi_{\mathrm{id}} \wedge \varphi_{\neg\sigma})\big)\Big)$$

where φ_{fd}, φ_{id}, and $\varphi_{\neg\sigma}$ will be defined below, but we explain their mission now. For each functional dependency $\mathrm{fd} \in \Gamma$, the formula φ_{fd} will hold on a transition $t = (\mathrm{I}, \mathrm{AC}, \mathrm{I}')$ exactly when fd holds on the restriction of I' to the schema predicates from Sch, and similarly for φ_{id}. The formula $\varphi_{\neg\sigma}$ checks that I' does not satisfy the functional dependency σ. Thus this formula will imply that the configuration is a witness showing that Γ does not imply σ.

We now explain how the different formulas are built. Let $\mathrm{fd} = R : P \to p$ where P are positions of relation R and p is a position of R. The formula φ_{fd} will be:

$$\mathsf{AX}\Big(\exists\vec{x}\vec{y}\ \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{x},\vec{y}) \wedge \bigwedge_{i\in P} x_i = y_i\ \wedge R_{\mathrm{post}}(\vec{x}) \wedge R_{\mathrm{post}}(\vec{y})$$
$$\Rightarrow \exists\vec{x}'\vec{y}\ \ \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{x}',\vec{y}') \wedge x'_p = y'_p\Big)$$

Here we use the derived "box" modality $\mathsf{AX}\varphi = \neg\mathsf{EX}\neg\varphi$. Note that φ_{fd} occurs in formula $\psi(\Gamma,\sigma)$ in a context where we know that only accesses to R_i have been done – hence only in contexts where $\mathrm{Chk}^{\mathrm{FD}}(R)$ must be empty. Since the only access methods for the relations $\mathrm{Chk}^{\mathrm{FD}}(R)$ are boolean, this means that after one transition we can have at most one tuple in $\mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{x},\vec{y})$. Thus doing a modality AX followed by a test that $\mathrm{Chk}^{\mathrm{FD}}(R)(\vec{x},\vec{y}) \wedge R_{\mathrm{post}}(\vec{x}) \wedge R_{\mathrm{post}}(\vec{y})$ holds amounts to testing an arbitrary pair $\vec{x},\vec{y}$ satisfying R prior to the access. The formula thus asserts that for any such pair of tuples in R, if they agree on all positions in the source of the FD, they agree on the target of the FD.

We can use a similar trick with the formula $\varphi_{\neg\sigma}$:

$$\mathsf{EX}\Big(\exists\vec{x}\vec{y}\ \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{x},\vec{y}) \wedge \bigwedge_{i\in P} x_i = y_i \wedge$$
$$R_{\mathrm{post}}(\vec{x}) \wedge R_{\mathrm{post}}(\vec{y}) \wedge$$
$$\neg\exists\vec{x}'\vec{y}\ \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{x}',\vec{y}') \wedge x'_p = y'_p\Big)$$

Now fix an id $R[A_1,\cdots,A_n] \subseteq S[B_1,\cdots,B_n]$, and we define φ_{id} to be

$$\mathsf{AX}\Big(\ \mathrm{IsBind}_{\mathrm{CheckIncDep}(R)} \wedge R_{\mathrm{post}}(\vec{x}) \wedge$$
$$\exists\vec{x}\ \ \mathrm{CheckIncDep}(R)_{\mathrm{post}}(\vec{x}) \quad \Longrightarrow$$
$$\mathsf{EX}\big(\ \mathrm{IsBind}_{\mathrm{CheckIncDep}(S)} \wedge \exists\vec{x}\ \ \mathrm{CheckIncDep}(R)_{\mathrm{post}}(\vec{x}) \wedge$$
$$\exists\vec{y}\,\mathrm{CheckIncDep}(S)_{\mathrm{post}}(\vec{y})\ \wedge \bigwedge_{i\le n} x_{A_i} = y_{B_i}\big)\Big)$$

This states that whenever we do a "test access" that returns an element of R, there is some access we can do immediately afterwards in the LTS that reveals a matching tuple in S. As

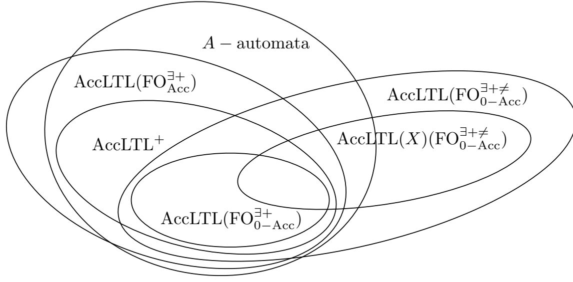

Figure 2: Inclusions between language classes.

in the case of φ_{fd} above, the accesses we perform are boolean, and hence cannot be creating any new elements of S – thus the revealed match must have been in the configuration prior to the access. □

6. CONCLUSIONS AND RELATED WORK

In this work we introduced the notion of querying the access paths that are allowed by a schema. We presented decidable specification languages for doing this, and gave undecidability results showing several limits of such languages. Figure 2 shows the inclusions of the languages considered in the paper, excluding those for branching time. All of the containments shown in the diagram are straightforward. The containment of $\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$ in AccLTL^{+} does require one to deal with the fact that $\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}}$ sentences are not required to be binding-positive. The inclusion follows by first rewriting negated 0-ary $\mathrm{IsBind}_{\mathrm{AcM}}$ predicates using the rule $\mathrm{IsBind}_{\mathrm{AcM}} = \bigvee_{\mathrm{AcM}' \neq \mathrm{AcM}} \mathrm{IsBind}_{\mathrm{AcM}'}$, then replacing the 0-ary predicate by existentially-quantified n-ary predicates.

All the inclusions in the diagram also turn out to be strict. We omit the proofs for this, which use standard techniques: e.g. A-automata can express parity conditions on the length of paths, which first-order languages like AccLTL^{+}, or even AccLTL($\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$), can not do.

Table 1 shows the complexity of satisfiability for each specification formalism, along with application examples. DjC indicates that the language can express relevance of an access in the presence of disjointness constraints, while FD, DF, AccOr refer to functional dependencies, dataflow restrictions, and access order restrictions, respectively.

Our work leaves open a number of questions concerning the logics we study – for example, we leave open the exact complexity of AccLTL^{+}, which lies between double- and triple- exponential time. We also do not have tight bounds for our more restricted fragments (e.g. with only the 0-ary version of $\mathrm{IsBind}_{\mathrm{AcM}}$) in the important case of grounded access paths.

Although this is, to the best of our knowledge, the first work on languages for describing access paths through a schema with binding patterns, there is a strong formal connection to work on verifying data-driven services, as well as other work in the area of hidden Web querying. We review the closest connections below.

Data-driven services. Our work is closely related to a line of research on relational transducers and models for data-driven services, beginning with Abiteboul et. al.'s [2], and continuing through work of Spielmann [21], Deutsch, Su and Vianu (e.g. [14]), Fritz et. al. [16], and Deutsch et. al. [12]. All of these works deal with specification languages for transition systems in which transitions may involve the consuming of relational inputs from an external environment, the production of output tuples, and the modification of internal state (perhaps in the form of an additional relational store). In our application, we talk of accesses rather than inputs from an environment, with a response consisting of revealing a hidden database instance, rather than updating an internal store. But in the results of this paper, one can just as easily think of identifying the hidden Web database with an internal store, with the accesses being non-deterministic inserts into the store.

Nevertheless, the logics that arise naturally in our setting appear orthogonal to those studied in prior work. The initial Abiteboul et. al. paper [2] focused on "Spocus transducers" (semi-positive output and cumulative state) which take full relational inputs, with their internal relations only accumulating them. A direct comparison with our model is difficult, since we do not have a notion of "output" – but if we restrict Spocus transducers to boolean output and singleton inputs, they are not as powerful as our model, since in our case the internal state can be modified in non-trivial ways. [2] proves an undecidability result for an extension of Spocus transducers in which the inserted data is allowed to be a projection of the "input relations" (Prop. 3.1 of [2]). The technique applied is similar to that in Theorem 5.2, but projection is orthogonal to the update given by access methods. In our terms, this extension would amount to having the information added to the hidden database be a projection of the accessed relations. On the other hand, the addition of projection does not give the ability to model access methods, which restrict the input relations by requiring them to satisfy a selection criterion.

Later works [21, 14, 12, 16] deal with transducers that can delete as well as insert into their internal state. A key restriction is *input-guardedness*, which insures decidability [14] – input guardedness requires quantifications to be restricted to tuples generated from the environmental inputs. The analogous restriction in our setting would be to restrict quantification to the bindings, which would be much weaker than the logics we consider. Thus our decidability and complexity results are not subsumed by these works. On the other hand, guarded quantification over relational inputs is not supported by our logics, and hence we do not claim to subsume results in these works. In addition, [12] allows a built-in linear order on the domain, which we do not consider for our largest logics. Later work by Damaggio et. al. considers even richer signatures, including arithmetic [11].

Hidden Web querying. Our work is directly inspired by previous results on static analysis of schemas with limited access patterns, a line of work tracing back (at least) to Ullman's work [22] and Rajaraman et. al. [20], continuing with Chang/Li's work in the early 2000s [18, 17] Ludäscher/Nash's and Deutsch et. al.'s work in the mid-2000's [19, 13] and Cali et. al. [5]. All of them deal in one way or another with what sequences can occur within a sequence with limited access patterns. For example, the question of whether a query can always be answered using exact grounded access paths – the focus of most of these works above – can be expressed as a property of the LTS. Exact complexity bounds for query answering derived from the works above. Containment under access patterns has also been studied, particularly in [5], which establishes a coN-

Language	Complexity	DjC	FD	DF	AccOr
AccLTL($FO^{\exists+,\neq}_{Acc}$)	undecidable	Yes	Yes	Yes	Yes.
AccLTL($FO^{\exists+}_{Acc}$)	undecidable	Yes	No	Yes	Yes
$AccLTL^{+}$	in 3EXPTIME	Yes	No	Yes	Yes
A-automata	2EXPTIME-compl.	Yes	No	Yes	Yes
AccLTL($FO^{\exists+}_{0-Acc}$)	PSPACE-compl.	Yes	No	No	Yes
AccLTL($FO^{\exists+,\neq}_{0-Acc}$)	PSPACE-compl.	Yes	Yes	No	Yes
AccLTL(X)($FO^{\exists+,\neq}_{0-Acc}$)	Σ_2^P-compl.	Yes	Yes	No	No

Table 1: Complexity and application examples for path specifications.

EXPTIME upper bound for conjunctive queries. [3] proves a matching coNEXPTIME lower bound for containment for conjunctive queries, and a co-2NEXPTIME upper bound for positive queries. [3] also defines the notion of long-term relevance (LTR). They prove a Σ_2^p-completeness result for LTR over general access paths ("independent accesses", in their terminology) while providing a NEXPTIME-completeness result for conjunctive queries and a 2NEXPTIME bound for LTR of positive queries over grounded accesses paths ("dependent accesses").

Our work provides a general framework where we can express properties of access paths, including containment, LTR, their combinations, and their restrictions to constraints. By providing these within a boolean closed logic, we give a flexible means of combining properties that one wishes to verify. Our 2EXPTIME result for non-emptiness of A-automata gives a bound on containment under access patterns and long-term relevance, as mentioned in the discussion after Theorem 4.6. This is better than the prior bounds from [5, 3].

Note that [3] also makes some erroneous claims: 1. A co2NEXPTIME lower bound for containment of positive queries under access patterns, which is at odds (relative to complexity-theoretic hypothesis) with our 2EXPTIME upper bound 2. A coNEXPTIME upper bound for containment of UCQs under general access patterns. The proof given there only works for schemas with a single-access per relation, while in subsequent work, we have shown that the problem is 2EXPTIME hard if the single-access restriction is dropped.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[3] M. Benedikt, G. Gottlob, and P. Senellart. Determining relevance of accesses at runtime. In *PODS*, 2011.

[4] A. Calì, D. Calvanese, and D. Martinenghi. Dynamic query optimization under access limitations and dependencies. *J. UCS*, 15(1):33–62, 2009.

[5] A. Calì and D. Martinenghi. Conjunctive query containment under access limitations. In *ER*, 2008.

[6] A. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[7] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive Datalog programs. *JCSS*, 54(1):61–78, 1997.

[8] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *JCSS*, 54(1):61–78, 1997.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[10] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 1997.

[11] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. In *ICDT*, 2011.

[12] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, 2009.

[13] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.

[14] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *JCSS*, 73:442–474, May 2007.

[15] E. Emerson. Temporal and modal logic. In *Handbook of Th. Comp. Sci.*, volume B. MIT, 1990.

[16] C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *ICDT*, 2009.

[17] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3):211–227, 2003.

[18] C. Li and E. Y. Chang. Answering queries with useful bindings. *ACM TODS*, 26(3):313–343, 2001.

[19] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[20] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[21] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS*, 66(1):40–65, 2003.

[22] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, V2*. Comp. Sci. Press, 1989.

[23] M. Y. Vardi. Alternating automata and program

verification. In *Computer Science Today*, pages 471–485, 1995.

APPENDIX

A. PROOF DETAILS

A.1 Proof of Theorem 3.1

Recall the statement:

> The satisfiability problem for $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ is undecidable

The proof works by reducing the problem of determining whether a collection Γ of functional dependencies (fds) and inclusion dependencies (ids) implies another functional dependency σ. Since this problem is known to be undecidable [6], it suffices to reduce it to unsatisfiability of a $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formula.

We now explain the details of the construction. We assume all positions carry the same type (since the undecidability of implication for IDs and FDs is in the untyped setting). We will also verify the reduction over instances where all relations are nonempty, allowing us to avoid certain corner cases.

We first give the schema, which extends the relational schema Sch for the dependencies. For each relation R of arity k, we have a relation $\mathrm{Succ}(R)$ of arity $2k$: informally, $\mathrm{Succ}(R)$ will be the successor relation referred to above. There are two relations $\mathrm{Beg}(R)$ (with boolean access $\mathrm{IsBind}_{\mathrm{Beg}(R)}$) and $\mathrm{End}(R)$ (having boolean access $\mathrm{IsBind}_{\mathrm{End}(R)}$) with the same arity as R; these will store the minimal and the maximal tuples for the ordering generated by $\mathrm{Succ}(R)$. In addition there are relations $\mathrm{CheckIncDep}(R)$ and with the same arity as R, having boolean accesses $\mathrm{IsBind}_{\mathrm{CheckIncDep}(R)}$. These are used to check the inclusions dependencies for R. Similarly we will have predicate $\mathrm{Chk}^{\mathrm{FD}}(R)$ with arity twice that of R.

We have a subformula that describes a path that fills the relations $\mathrm{Beg}(R), \mathrm{Succ}(R), \mathrm{End}(R)$. The following formula $\varphi_{next(R)}$ describes a transition adding a tuple $(\vec{t}_1, \vec{t}_2)$ to $\mathrm{Succ}(R)$ such that (i) $\vec{t}_1$ has a predecessor in the current instance but does not have a successor in the current instance and (ii) $\vec{t}_2$ does not appear in the successor relation at all.

$$
\begin{aligned}
&(\exists \vec{t}_1 \vec{t}_2 \vec{t}_3 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t}_3, \vec{t}_1)) \wedge \\
&(\exists \vec{t}_1 \vec{t}_2 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Succ}(R)_{\mathrm{post}}(\vec{t}_1, \vec{t}_2)) \wedge \\
\neg&(\exists \vec{t}_1 \vec{t}_2 \vec{t}_3 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t}_1, \vec{t}_3)) \wedge \\
\neg&(\exists \vec{t}_1 \vec{t}_2 \vec{t}_3 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \\
&\mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t}_2, \vec{t}_3) \vee \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t}_3, \vec{t}_2))
\end{aligned}
$$

The first subformula checks the first part of (i), the second ensures the second part of (i) and the third subformula checks (ii). The correctness of this formula relies on the fact that there is only one tuple in $\mathrm{Succ}(R)$.

There are also two formulas $\varphi_{\mathrm{Beg}(R)}$ and $\varphi_{\mathrm{End}(R)}$ that mark the start and end of the creation of the successor relation. $\varphi_{\mathrm{Beg}(R)}$ is defined as:

$$
\begin{aligned}
&\exists \vec{t}_1 \;\; \mathrm{IsBind}_{\mathrm{Beg}(R)}(\vec{t}_1) \wedge \mathrm{Beg}(R)_{\mathrm{post}}(\vec{t}_1) \wedge \\
&\mathsf{X}\big(\exists \vec{t}_1 \vec{t}_2 \,.\, \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Beg}(R)_{\mathrm{pre}}(\vec{t}_1) \wedge \\
&\exists \vec{t}_1 \vec{t}_2 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Succ}(R)_{\mathrm{post}}(\vec{t}_1, \vec{t}_2) \wedge \\
&\neg(\exists \vec{t}_1 \vec{t}_2 \;\; \mathrm{IsBind}_{\mathrm{Succ}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Beg}(R)_{\mathrm{pre}}(\vec{t}_2))\big)
\end{aligned}
$$

where $\varphi_{\mathrm{End}(R)}$ is defined as:

$$\exists \vec{t_1}\ \ \mathrm{IsBind}_{\mathrm{End}(R)}(\vec{t_1}) \wedge \exists \vec{t_3}\, \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t_3}, \vec{t_1}) \wedge$$
$$\exists \vec{t_1}\ \ \mathrm{IsBind}_{\mathrm{End}(R)}(\vec{t_1}) \wedge \mathrm{End}(R)_{\mathrm{post}}(\vec{t_1}) \wedge$$
$$\neg(\exists \vec{t_1}\vec{t_2}\ \ \mathrm{IsBind}_{\mathrm{End}(R)}(\vec{t_1}) \wedge \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t_1}, \vec{t_2}))$$

Then the formula we ultimately create will include subformulas of the form:

$$\varphi_{\mathrm{Beg}(R)} \wedge \mathsf{X}(\varphi_{next(R)} \mathsf{U}\, (\varphi_{\mathrm{End}(R)} \wedge \mathsf{X}\varphi_{\mathrm{verify}}))$$

where $\varphi_{\mathrm{verify}}$, defined next, checks the constraints on R. Note that in any path that satisfies such a formula, the instance associated to the position satisfying $\varphi_{\mathrm{verify}}$ must interpreted $\mathrm{Succ}(R)$ as the successor relation of a linear order on a collection of tuples whose arity agrees with R, with first element the sole tuple in $\mathrm{Beg}(R)$ and last element the sole tuple in $\mathrm{End}(R)$.

Check of the functional dependencies. Recall that in the schema we have relations $\mathrm{Chk}^{\mathrm{FD}}(R)$ with arity twice that of R. These are used to check if every pair $(\vec{t_1}, \vec{t_2})$ satisfies all FDs in Γ pertaining to R. At the end of the check, the FD is satisfied iff $\mathrm{Chk}^{\mathrm{FD}}(R)$ contains all tuples from the part of the instance generated by $\mathrm{Succ}(R)$.

First we present a sentence $\varphi_{\text{fd-tuple}}$ which checks that the current binding is a pair $(\vec{t_1}, \vec{t_2})$ which is not a counterexample to an $FD\ fd = A \longrightarrow B$ on R:

$$\exists \vec{t_1}\vec{t_2}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge$$
$$\neg(\exists \vec{t_1}\vec{t_2}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge \Pi_A(\vec{t_1}) = \Pi_A(\vec{t_2})) \vee$$
$$\exists \vec{t_1}\vec{t_2}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge \Pi_B(\vec{t_1}) = \Pi_B(\vec{t_2})$$

We now consider an iterator $\varphi_{\mathrm{checknext}}$ that checks all pairs. The formula $\varphi_{FD-init}$ begins the check for the first pair in R:

$$\exists \vec{t_1}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_1}) \wedge \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{t_1}, \vec{t_1}) \wedge$$
$$\exists \vec{t_1}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_1}) \wedge \mathrm{Beg}(R)_{\mathrm{pre}}(\vec{t_1})$$

This holds at a position iff it has an access on a pair $(\vec{t_1}, \vec{t_1})$ where $\vec{t_1}$ is the first tuple in the ordering, and the access is successful. Note that there is no need to check the FD on such a tuple.

The formula $\varphi_{FD-next}$ performs the iteration:

$$\exists \vec{t_1}\vec{t_2}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge$$
$$\mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{t_1}, \vec{t_2}) \wedge$$
$$(\neg\exists \vec{t_1}\vec{t_2}\, \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge$$
$$\mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{pre}}(\vec{t_1}, \vec{t_2})) \wedge$$
$$(\varphi_1 \vee \varphi_2) \wedge \varphi_{\text{fd-tuple}}$$

where φ_1 is

$$\exists \vec{t_1}\vec{t_2}\vec{t_3}\, \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R))}(\vec{t_1}, \vec{t_2})\ \ \wedge \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{pre}}(\vec{t_1}, \vec{t_3}) \wedge$$
$$\mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t_3}, \vec{t_2})$$

and φ_2 is

$$\exists \vec{t_1}\vec{t_2}\vec{t_3}\vec{t_4}\ \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R))}(\vec{t_1}, \vec{t_2})\ \ \wedge \mathrm{Beg}(R)_{\mathrm{pre}}(\vec{t_2})$$
$$\wedge\, \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{pre}}(\vec{t_3}, \vec{t_4})\, \mathrm{Succ}(R)_{\mathrm{pre}}(\vec{t_3}, \vec{t_1}) \wedge \mathrm{End}(R)_{\mathrm{pre}}(\vec{t_4})$$

The formula states that the current position has an access to a pair $(\vec{t_1}, \vec{t_2})$ that has not been checked before where the access passes the test, and where $Pred(\vec{t_1}, \vec{t_2})$ has been checked, where this denotes the predecessor in the lexicographic ordering: either $\vec{t_1}$ paired with the predecessor of $\vec{t_2}$ (as in φ_1) or the predecessor of $\vec{t_1}$ and the final tuple (φ_2).

The formula $\varphi_{\text{FD-end}}$ ends the check :

$$\exists \vec{t_1}\vec{t_2}\, \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t_1}, \vec{t_2}) \wedge \mathrm{End}_{\mathrm{pre}}(\vec{t_2}) \wedge \varphi_{\text{fd-tuple}}$$

The formula $\varphi_{\text{FD-block}} = \varphi_{\text{FD-init}} \wedge \mathsf{X}\varphi_{\text{FD-next}} U \varphi_{\text{FD-end}}$ thus will hold at a position iff there is a sequence of accesses after it verifying the FD for every pair $(\vec{t_1}, \vec{t_2})$.

The treatment of IDs and the treatment of the negation of FDs is similar (and is also spelled out in the proof of Theorem 5.2).

A.2 Details for the proof of Proposition 4.4

Recall the statement:

> Let Q and Q' be two positive queries, ACS a set of access methods, and Σ a set of disjointness constraints. One can efficiently produce an A-automaton A such that Q is contained in Q' under limited access patterns with disjointness constraints iff the language recognized by A is empty. A similar statement holds for long-term relevance under disjointness constraints.

Proof. We denote by Q_{post} and Q'_{post} the queries derived from Q and Q' by replacing any atom $R(x)$ in them by $R_{\mathrm{post}}(x)$. Let A be an A-automaton with two states s_0 and s_1. The initial state is s_0 and the final state is s_1. The transitions are

$$\delta(s_0, s_0) = \bigwedge_{\sigma\in\Sigma} \neg\varphi_\sigma \wedge \neg Q'_{\mathrm{post}}$$
$$\delta(s_0, s_1) = \bigwedge_{\sigma\in\Sigma} \neg\varphi_\sigma \wedge \neg Q'_{\mathrm{post}} \wedge Q_{\mathrm{post}}$$

It is easy to see that each instance I resulting from an access $(\mathrm{AcM}, \bar{b})$ to I′ satisfies Σ iff $(\mathrm{I}', (\mathrm{AcM}, \bar{b}), \mathrm{I})$ satisfies $\bigwedge_{\sigma\in\Sigma} \neg\varphi_\sigma$. The last instance resulting from a sequence of accesses ρ satisfies Q and $\neg Q'$ iff the transition of ρ satisfies $\neg Q'_{\mathrm{post}} \wedge Q_{\mathrm{post}}$.

Now note that any disjointness constraint σ between R and S can be expressed by asserting that the negation of the following conjunctive query holds initially and also on every transition:

$$\varphi_\sigma = \exists x, y R_{\mathrm{post}}(x) \wedge S_{\mathrm{post}}(y).$$

□

A.3 Proof of Lemma 4.5

Recall the statement

> For every AccLTL^+ formula φ there is an equivalent A-automaton A of size exponential in the size of φ

Proof. Let an AccLTL^+ formula φ be given. We will translate φ into an A-automaton in several steps. In the first step we translate φ into an LTL formula over a finite set of predicates. To do this, we consider an alphabet Φ that contains one unary predicate P_χ for every formula χ that is a maximal $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ subformula of φ. We denote by $\tilde{\varphi}$ the LTL formula obtained from φ by replacing each χ in Φ by P_χ. We also transform an access path $p = (\mathrm{I}_1, \mathrm{AC}_1, \mathrm{I}'_1), \ldots, (\mathrm{I}_n, \mathrm{AC}_n, \mathrm{I}'_n)$ into a word $\tilde{p} = S_1, \ldots, S_n$

over the finite alphabet 2^{Φ} where S_i is the set of all formulas in Φ that hold on $(\mathrm{I}_i, \mathrm{AC}_i, \mathrm{I}'_i)$, $i \leq n$. With these definitions, the following proposition is obvious:

PROPOSITION A.1. *For every* AccLTL$^+$ *formula* φ *and every access path* p, φ *holds on* p *iff* $\tilde{\varphi}$ *holds on* $\tilde{p}$.

We next show that the LTL formula $\tilde{\varphi}$ constructed by the above translation has a certain monotonicity property. We will call a propositional symbol $P \in \mathrm{Prop}$ *even* in $\tilde{\varphi}$ if each occurrence of P in $\tilde{\varphi}$ is under an even number of negations. We will later exploit the fact that, by definition of AccLTL$^+$, every atom $\mathrm{IsBind}_{\mathrm{AcM}}$ must occur under an even number of negations in φ. Hence if χ contains $\mathrm{IsBind}_{\mathrm{AcM}}$ then P_χ is even in $\tilde{\varphi}$.

Given a word $w = S_1, \ldots, S_n$, a position $i \leq n$ and a proposition $P \in \mathrm{Prop}$, we define $w_{P,i} = S_1, \ldots, S_{i-1}, S_i \cup \{P\}, S_{i+1}, \ldots, S_n$.

PROPOSITION A.2. *Let* φ *be an LTL formula and let* P *be an even propositional symbol in* φ. *Then for all words* w *over* 2^{Prop} *and positions* $i \leq n$, *if* w *is a model of* φ *then* $w_{P,i}$ *is a model of* φ.

PROOF. We show for all words w and positions i, j in w, that if (w, j) is a model of φ then $(w_{P,i}, j)$ is a model of φ. We show this statement by induction on φ. Fix some word $w = S_1 \ldots S_n$, some $i, j \leq n$ and assume that $w, j \models \varphi$. If φ is a propositional symbol P that occurs in S_i then the statement is clearly true. If P does not occur in S_i then i must be distinct from j and the statement is trivial. The case that φ is of the form $\neg P$ is not possible because otherwise P would not be even. The cases where φ is of the form $\mathsf{X}\psi$, $\mathsf{F}\psi$ of $\psi \mathsf{U} \psi'$ follow from the induction hypothesis. □

In the second step, we translate the LTL formula $\tilde{\varphi}$ into an equivalent alternating finite automaton $A_{\tilde{\varphi}}$ over the alphabet $\Sigma = 2^{\Phi}$. We adapt a construction from [23] to finite words. Recall that an *alternating finite automaton* is a tuple $(Q, \mathrm{Prop}, q_0, \rho, F)$ where Q is a finite set of states, Prop is a finite set of propositions, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of accepting states, and ρ is a function mapping $Q \times 2^{\mathrm{Prop}}$ to positive boolean combinations of states. A *run* over a word $w = S_1, \ldots, S_n$ with $S_i \in 2^{\mathrm{Prop}}$ is a finite Q-labelled tree whose root is labelled by q_0 and which satisfies the following property: If x is a node at level i that is labelled q, then x has k children $x_1, \ldots, x_k$, $k \leq |Q|$ such that $\{x_1, \ldots, x_k\}$ is a model of $\rho(q, S_{i+1})$. A run is *accepting* if each state that labels a leaf is in F. Note that in an accepting run we cannot have $\rho(s, q) = \mathrm{false}$, because "false" is not satisfiable.

Given a finite set of propositions Prop, an LTL formula φ over the alphabet Prop, we define an alternating automaton A_φ that has a state for each subformula of $\tilde{\varphi}$. We define the dual $\overline{\varphi}$ of a formula φ as follows:

$$\begin{aligned}
\overline{P} &= \neg P \quad \text{for } P \in \mathrm{Prop} \\
\overline{\mathrm{true}} &= \mathrm{false} \\
\overline{\mathrm{false}} &= \mathrm{true} \\
\overline{\varphi \wedge \psi} &= \overline{\varphi} \vee \overline{\psi} \\
\overline{\varphi \vee \psi} &= \overline{\varphi} \wedge \overline{\psi}
\end{aligned}$$

We define the transition function $\rho : Q \times 2^{\mathrm{Prop}} \to Q$ as follows:

$$\begin{aligned}
\rho(P, S) &= \mathrm{true} && \text{if } P \in S \\
\rho(P, S) &= \mathrm{false} && \text{if } P \notin S \\
\rho(\neg P, S) &= \mathrm{true} && \text{if } P \notin S \\
\rho(\neg P, S) &= \mathrm{false} && \text{if } P \in S \\
\rho(\varphi \wedge \psi, S) &= \rho(\varphi, S) \wedge \rho(\psi, S) \\
\rho(\neg\varphi, S) &= \overline{\rho(\varphi, S)} \\
\rho(\mathsf{X}\varphi, S) &= \varphi \\
\rho(\varphi \mathsf{U} \psi, S) &= \rho(\psi, S) \vee (\rho(\varphi, S) \wedge \varphi \mathsf{U} \psi)
\end{aligned}$$

The initial state of $A_{\tilde{\varphi}}$ is $\tilde{\varphi}$ and any formula of the form $\neg(\varphi \mathsf{U} \psi)$ and $\neg \mathsf{X}\varphi$ are final states. A straightforward proof by induction shows that $L(\varphi) = L(A_\varphi)$.

Next, we turn $A_{\tilde{\varphi}}$ into an equivalent non-deterministic automaton $B_{\tilde{\varphi}}$ using a standard subset construction. Note that the transitions of $B_{\tilde{\varphi}}$ are labelled by sets of propositions. We will now consider a model of automata, called *boolean automata*, in which the transitions are labelled by boolean combinations of propositions. If such an automaton is in state p and the current input symbol is a set of propositions S, then it can transition to a state q if there is a transition (p, φ, q) such that S is a model of φ. Note that every non-deterministic finite automaton A over the alphabet 2^{Prop} is equivalent to a boolean automaton B: the boolean automaton B obtained from A by replacing each set S on a transition by the formula $\psi^- \wedge \psi^+$ where

$$\psi^- = \bigwedge_{P \notin S} \neg P \qquad \psi^+ = \bigwedge_{P \in S} P$$

PROPOSITION A.3. *Let* B *be a boolean automaton that is obtained from an LTL formula* ψ *using the construction described above. Then* B *is equivalent to a boolean automaton* C *in which no even propositional symbol of* ψ *occurs negated.*

PROOF. Assume that P is even in ψ and that B contains a transition $d = (s, \varphi \wedge \neg P, s')$. We claim that if D is the automaton obtained from B by adding a transition $d' = (s, \varphi \wedge P, s')$ then $L(B) = L(D)$. It is clear that by adding a transition the accepted language can only increase, so we only need to show that $L(D) \subseteq L(B)$. Assume towards a contradiction that there is a word $w \in L(D)$ that is not in $L(B)$. Clearly, the accepting run of w in D must make use of the added transition d'. Let $i_1, \ldots, i_n$ be all the positions on w in which this accepting run uses the transition d' of D. Consider the word w' that is obtained from w by removing the symbol P from every position i_j, $j \leq n$. Clearly this word w' must be accepted by B. It follows from Proposition A.2 that in fact w is accepted by B. This is a contradiction of the assumption that $w \notin L(B)$, hence $L(C) = L(B)$. Note also that if we replace the two transitions d and d' by a single transition (s, φ, s') then the accepted language does not change, simply because $(\varphi \wedge \neg P) \vee (\varphi \wedge P)$ is equivalent to φ. The claim follows because we can eliminate each transition on which an propositional symbol occurs negated that is even in φ. □

We use Proposition A.3 to transform $B_{\tilde{\varphi}}$ into a boolean automaton $C_{\tilde{\varphi}}$ in which no proposition that is even in $\tilde{\varphi}$ occurs negated. In the final step we will translate $C_{\tilde{\varphi}}$ into

an A-automaton A as follows. Recall that the boolean formulas on the transitions of $C_{\tilde{\varphi}}$ are over propositions of the form P_χ where χ is a maximal $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ subformula of φ, the formula we were given at the beginning of this proof. Let D be the automaton obtained from $C_{\tilde{\varphi}}$ by replacing each proposition P_χ by χ. To verify that D is an A-automaton, we must make sure that no predicate IsBind AcM occurs in an $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula that is negated. This is the case because, as we observed previously, if $\mathrm{IsBind}_{\mathrm{AcM}}$ occurs in an $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula χ of φ, then P_χ is even in $\tilde{\varphi}$. In this case all negated occurrences of P_χ have been removed when $B_{\tilde{\varphi}}$ was translated into $D_{\tilde{\varphi}}$. It follows that D is an A-automaton.

We still need to verify that D has size at most exponential in φ. This is the case because the only blow-up in the above construction happens when we translate the AFA $A_{\tilde{\varphi}}$ into the NFA $B_{\tilde{\varphi}}$. In particular the translation from $\tilde{\varphi}$ to $A_{\tilde{\varphi}}$ does not introduce a blowup.

A.4 Proof of Lemma 4.9

Recall the statement:

> For every A-automaton A, there are progressive A-automata $A_1, \ldots, A_n$, such that, for each $i \leq n$, the size of A_i is polynomial in the size of A, n is exponential in the size of A, and $L(A)$ is empty iff $L(A_1) \cup \ldots \cup L(A_n)$ is empty.

Proof. The proof consists of two steps. We define an *almost progressive A-automaton* to be an A-automaton that satisfies every property of the definition of a progressive A-automaton apart from Property 3, which concerns transitions leaving the initial state. In the first step we show that every A-automaton can be translated into an exponential number of almost progressive A-automata. This will be the subject of Claim 1. In the second step, we show that every almost progressive A-automaton can be translated into an exponential number of A-automata. This will be the content of Claim 2.

Claim 1. *For every A-automaton A, there are almost progressive A-automata $A_1, \ldots, A_n$, such that, for each $i \leq n$, the size of A_i is polynomial in the size of A, n is exponential in the size of A, and $L(A)$ is empty iff $L(A_1) \cup \ldots \cup L(A_n)$ is empty.*

Proof of Claim 1. Assume we are given an A-automaton A consisting of (Q, q_0, δ, F). Recall that the transitions of A are of the form $(s, \psi^- \wedge \psi^+, s')$ where ψ^- is a positive boolean combination of negated $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences that can not contain the predicate IsBind, and where ψ^+ is a positive boolean combination of $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences. In the following we will say that a formula is in *transition normal form* (TNF) if it is of this form. Note that a formula in this normal form might contain several atoms with a predicate $\mathrm{IsBind}_{\mathrm{AcM}}$. Condition 1 in the definition of a progressive automaton requires that that if $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ and $\mathrm{IsBind}_{\mathrm{AcM}'}(\bar{t}')$ are atoms in a formula on a transition of A, then $\mathrm{AcM} = \mathrm{AcM}'$. We will argue that any formula φ in TNF is equivalent to a disjunction of formulas that satisfy Condition 1. We let φ_{AcM} be the formula obtained from φ by replacing each atom with predicate $\mathrm{IsBind}_{\mathrm{AcM}'}$ by false whenever $\mathrm{AcM}' \neq \mathrm{AcM}$. Recall that at any point in an access path, the predicate $\mathrm{IsBind}_{\mathrm{AcM}_i}$ holds of exactly one tuple while all other predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ are empty. Hence, when evaluated over access paths, φ is equivalent to the disjunction of formulas φ_{AcM} over all accesses AcM used in A. We will perform this substitution on all formulas in transitions of A to obtain an automaton in which all transitions are in the form required by Condition 1 in the definition of a progressive automaton. In the following we will assume that A is already normalized in this way.

We let Φ be the set of all maximal $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formulas that appear as subformulas of formulas on transitions of A (hence Φ contains only positive formulas). Let $\tilde{\Phi}$ be the set of formulas $\tilde{\varphi}$ with $\varphi \in \Phi$, where $\tilde{\varphi}$ is as defined above Definition 4.8. Given a subset S of $\tilde{\Phi}$, we let the *complete type of* S, denoted φ_S, be the conjunction that contains every element of S as a positive atom and each element of $\tilde{\Phi} \setminus S$ as a negative atom. We will call a sequence $\bar{S} = S_1, \ldots S_h$ of subsets of $\tilde{\Phi}$ with $S_i \subsetneq S_{i+1}$ a *type evolution*.

We now fix a type evolution $\bar{S} = S_1, \ldots, S_h$ of length $h \leq |\tilde{\Phi}|$, and $h-1$ transitions $\bar{d} = d_1, \ldots, d_{h-1}$ of A. We will construct an almost progressive automaton $A_{\bar{S},\bar{d}}$. The state set of $A_{\bar{S},\bar{d}}$ is $Q \times \bar{S}$. For each $i \leq h$, we add a transition

$$((s, S_i), \varphi_{S_i} \wedge \tilde{\varphi}_{S_i} \wedge \varphi^- \wedge \varphi^+, (s', S_i))$$

to $A_{\bar{S},\bar{d}}$ iff A contains a transition $(s, \varphi^- \wedge \varphi^+, s')$ such that φ_{S_i} implies $\tilde{\varphi}^- \wedge \tilde{\varphi}^+$. We will now add some transitions to $A_{\bar{S},\bar{d}}$ in order to connect states of the form (s, S) to states of the form (s', S') with $S \neq S'$ (note that such pairs of states are not yet connected). We will use the transitions in $\bar{d}$: For each $i < h$ we assume that d_i is of the form $(s, \varphi^- \wedge \varphi^+, s')$ and add the transition

$$((s, S_i), \varphi_{S_{i+1}} \wedge \varphi^- \wedge \varphi^+, (s', S_{i+1}))$$

to $A_{\bar{S},\bar{d}}$ iff $\tilde{\varphi}^- \wedge \tilde{\varphi}^+$ implies $\varphi_{S_{i+1}}$. We then choose (q_0, S_1) as the initial state of $A_{\bar{S},\bar{d}}$, all (q, S_h) with $q \in F$ as the final states, and we define $\Upsilon_{\mathrm{post}}((q, S)) = S$ for all states (q, S) of $A_{\bar{S},\bar{d}}$. It is straightforward to verify that all transitions satisfy Conditions 2, 4 and 6 of the definition of a progressive automaton.

It is possible that there is no path from the initial state to the final state of $A_{\bar{S},\bar{d}}$. As usual, we assume that $L(A_{\bar{S},\bar{d}}) = \emptyset$ in this case. With the above definitions it is easy to verify that

$$L(A) = \bigcup_{\bar{S} \in \mathcal{S},\ \bar{d} \in \delta^{h-1}} L(A_{\bar{S},\bar{d}})$$

where $\mathcal{S}$ is the set of type evolutions with respect to $\tilde{\Phi}$. The direction from left to right is obvious. For the other direction let w be an access path that is accepted by A. Consider the evaluation of the instance that A stores on the run that witnesses the acceptance of w. This evaluation of instances gives rise to a type evaluation $\bar{S}$. Let $\bar{d}$ be the sequence of transitions that A uses when moving from one type in $\bar{S}$ to the next. Then w is accepted by $A_{\bar{S},\bar{d}}$.

Note that $A_{\bar{S},\bar{d}}$ may not be an almost progressive automaton, because Condition 5 of the definition of a progressive automaton is not satisfied. There might be two problems: first, $A_{\bar{S},\bar{d}}$ might not be a sequence of maximal strongly connected components but rather have a DAG structure. This problem may be remedied by replacing an $A_{\bar{S},\bar{d}}$ by the set $A_{\bar{S},\bar{d},1}, \ldots, A_{\bar{S},\bar{d},m}$ of maximal subautomata of $A_{\bar{S},\bar{d}}$ whose strongly connected components form a linear order. It is obvious that $L(A_{\bar{S},\bar{d}}) = L(A_{\bar{S},\bar{d},1}) \cup \ldots \cup L(A_{\bar{S},\bar{d},m})$.

The second problem is that the formulas on transitions that connect strongly connected components of A might not

be of the correct form. In particular, they might not satisfy Condition 5 of the definition of a progressive automaton, which states that all atoms with a predicate $\mathrm{IsBind}_{\mathrm{AcM}}$ must be grounded. We address this problem by replacing each variable x in a transition between strongly connected components by a constant c. This replacement must replace different variables by different constants. Let A' be the result of replacing variables by constants in $A_{\bar{S},\bar{d},i}$ as described above. Clearly, this operation does not preserve the language accepted by the automaton. However, it is clear that $L(A_{\bar{S},\bar{d},i}) \supseteq L(A')$. In addition one can show that every access path p accepted by $A_{\bar{S},\bar{d},i}$ can be relabeled to an access path p' that is accepted by A'. It follows that $L(A_{\bar{S},\bar{d},i})$ is empty iff $L(A')$ is empty. With these modifications, the resulting automata are almost progressive automata.

It remains to show that at most exponentially many automata have been constructed. This is the case as there are at most exponentially many different automata $A_{\bar{S},\bar{d}}$ and there are at most exponentially many different linear orders that embed into a given DAG. This completes the proof of Claim 1. □

CLAIM 2. *For every almost progressive A-automaton A, there are progressive A-automata $A_1, \ldots, A_n$, such that, for each $i \leq n$, the size of A_i is polynomial in the size of A, n is exponential in the size of A, and $L(A)$ is empty iff $\bigcup_{\varphi,d,\bar{c}} L(A_{\varphi,d,\bar{c}})$ is empty.*

PROOF. Let $A = (S, s_0, F, \delta)$ be an almost progressive A-automaton. Then there exists a set Φ of pure post $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences, and a function Υ_{post} mapping the states of A to complete Φ-types. Recall that $\Upsilon_{\mathrm{post}}(s_0)$ is a conjunction $\varphi_1 \wedge \ldots \wedge \varphi_n$ of (possibly negated) pure post $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences in Φ. Let Φ_0 be the set complete Φ-types over a subset S_0 of $\{\varphi_1, \ldots, \varphi_n\}$. Let $\mathrm{Trans}(s_0)$ be the set of transitions of A that start in the initial state s_0. Let $\mathrm{arity}(\mathrm{Sch})$ be the maximal arity of a relation in schema of A, let Const be the union of all constants that appear in A with a set of $\mathrm{arity}(A)$ domain elements that do not appear as constants in A. We will define, for each $\varphi \in \Phi_0$, each $d \in \mathrm{Trans}(s_0)$, each $k \leq \mathrm{arity}(\mathrm{Sch})$, and each $\bar{c} \in \mathrm{Const}^k$ an A-automaton $A_{\varphi,d,\bar{c}}$.

Recall that as A is an almost progressive automaton, if a transition d contains two predicates IsBind AcM and $\mathrm{IsBind}_{\mathrm{AcM}'}$ then $\mathrm{AcM} = \mathrm{AcM}'$. Hence we can associate with every transition of A a unique access method, called the access method of that transition below.

Fix some $\varphi \in \Phi_0$, some $d \in \mathrm{Trans}(s_0)$, some $k \leq \mathrm{arity}(\mathrm{Sch})$, and some $\bar{c} \in \mathrm{Const}^k$. If the number of bound positions in the access of d is not k, then we define $A_{\varphi,d,\bar{c}}$ to be an A-automaton that accepts the empty language. Otherwise the state set of $A_{\varphi,d,\bar{c}}$ is $S \cup \{q_0\}$ for some state $q_0 \notin S$. We let q_0 be the initial state of $A_{\varphi,d,\bar{c}}$ and we let the accepting states of $A_{\varphi,d,\bar{c}}$ be the accepting states of A. The transitions of $A_{\varphi,d,\bar{c}}$ include all transitions of A (including d) and one extra transition that is obtained from d and φ as follows.

Let d be of the form (s_0, ψ, s) and let $\tilde{\varphi}$ be the formula obtained from φ by replacing all predicates R_{post} by R_{pre}. Let ψ' be obtained from ψ by replacing each predicate $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$, where $\bar{t}$ is a sequence of variables and constants, by $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{c})$ Then we add the transition $(q_0, \tilde{\varphi} \wedge \psi', s)$ to $A_{\varphi,d}$. We let $\Upsilon_{\mathrm{pre}}(s_0)$ be $\tilde{\varphi}$ (note that $\tilde{\varphi}$ can not contain a predicate $\mathrm{IsBind}_{\mathrm{AcM}}$). It is clear that $\tilde{\varphi} \wedge \psi'$ implies $\Upsilon_{\mathrm{pre}}(s_0)$, hence Condition 3 in the definition of a progressive A-automaton is satisfied. Still, $A_{\varphi,d,\bar{c}}$ might not be a progressive A-automaton, since s might be in the second strongly connected component of A, making all of the elements in the first SCC of A unreachable from q_0. In this case we remove all unreachable states. Then it is easy to check that the resulting automaton is a progressive automaton.

We still need to prove that $L(A)$ is empty iff $\bigcup_{\varphi,d,\bar{c}} L(A_{\varphi,d,\bar{c}})$ is empty. It is obvious that $L(A_{\varphi,d}) \subseteq L(A)$ for every φ and d. To show the inclusion of the left hand side in the right hand side, let p be some access path for which A has an accepting run r. Let d be the transition of A that r uses first and let $p_1 = (\mathrm{I}, (\mathrm{AcM}, \bar{b}), \mathrm{I}')$ be the first transition of p. Let $\tilde{p}_1$ be $(\mathrm{I}, (\mathrm{AcM}, \bar{b}), \mathrm{I})$ and let φ be the (unique) complete Φ-type that holds on $\tilde{p}_1$. Then it is easy to see that there is an access path p' that can be obtained from p by consistently renaming domain elements, such that $A_{\varphi,d,\bar{c}}$ has a run on p'. □

A.5 Translation from progressive A-automata non-emptiness to Datalog containment

We now give the construction of the Datalog program $\mathcal{P}_A$ and the positive query $\mathcal{P}'_A$ from a progressive automaton A, proving Lemma 4.10. Both the construction and the correctness proof to follow are quite involved.

Let a progressive A-automaton $A = (S, s_0, F, \delta)$ over schema Sch be given. As A is progressive, there must be a pure pre formula $\Upsilon_{\mathrm{pre}}(s_0)$, a set Φ of pure post $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences, and a function Υ_{post} mapping the states of A to complete Φ-types. Also, A must have a linearly-ordered component structure $C_1, \ldots, C_h$, $h = \mathrm{height}(A)$.

We will define a Datalog program $\mathcal{P}_A$ over the extensional schema $\mathrm{Sch}_{\mathrm{ext}}$ that contains for each $R \in \mathrm{Sch}$ and $1 \leq i \leq \mathrm{height}(A)$, a relation $\mathrm{BackgroundR}_i$, and for each $R \in \mathrm{Sch}$ and $0 \leq i \leq \mathrm{height}(A) - 1$ a relation $\mathrm{IntBackgroundR}_i$; the arity of both relations is equal to the the arity of R.

To define the intensional schema of $\mathcal{P}_A$, we need more notation. Let Φ^- (Φ^+ respectively) be the set of subformulas of formulas φ^- (φ^+ respectively) that appear on transitions $(s, \varphi^- \wedge \varphi^+, s')$ of A.

The intensional schema $\mathrm{Sch}_{\mathrm{int}}$ of $\mathcal{P}_A$ contains:

- for each $R \in \mathrm{Sch}$ and $1 \leq i \leq \mathrm{height}(A)$, the symbol ViewR_i with the arity of R.
- for every $k \leq m$, and every $1 \leq i \leq \mathrm{height}(A)$ where m is the highest arity of a relation in Sch, a symbol Val^i_k and a symbol $\mathrm{Val}^{\leq i}_k$, both of arity k.
- for every $1 \leq i \leq \mathrm{height}(A)$, a predicate ReachC_i of arity 0.
- for every subformula ψ of $\Upsilon_{\mathrm{pre}}(s_0)$ a predicate $\mathrm{IniEval}^{\psi}$ of arity f where f is the number of free variables of ψ.
- for every formula ψ in Φ^+, every $1 \leq i \leq \mathrm{height}(A)$, and every access AcM of Sch, a predicate $\mathrm{Eval}^{\psi}_{i,\mathrm{AcM}}$, of arity $f + b$, where f is the number of free variables of ψ and b is the number of input positions of AcM
- for every formula ψ in Φ^+, every $1 \leq i \leq \mathrm{height}(A)$, a predicate $\mathrm{IntEval}^{\psi}_i$ of arity $f + b$ where f is the number of free variables of ψ and b is the number of input positions of the access used in the unique transition from C_i to C_{i+1}.
- for every $1 \leq i \leq \mathrm{height}(A)$ there is a symbol Equal_i of arity 2.

The goal predicate of $\mathcal{P}_A$ is ReachC_h where $h = \mathrm{height}(A)$.

The intuition is that the relation R in the background database that the A-automaton A accesses is equal to the union of the relations $\text{BackgroundR}_1, \ldots, \text{BackgroundR}_h$, $h = \text{height}(A)$, along with the relations $\text{IntBackgroundR}_0, \ldots, \text{IntBackgroundR}_{h-1}$. The tuples in BackgroundR_i, $i \leq h$ are those that the background database might return when A accesses relation R whilst remaining in the same strongly connected component C_i. The tuples in IntBackgroundR_i are those that the background database returns when A accesses relation R whilst crossing from C_i into component C_{i+1}. The union of the relations $\text{ViewR}_1, \ldots, \text{ViewR}_i$ corresponds to the relation R in the instance that A stores internally when it leaves C_i. The predicate ReachC_i indicates that there is a path p of LTS associated with the schema of A, and a run r of A that starts in the initial state and ends in a state of C_i such that r is a run of A on p. The relation Equal_i represents the equality relation for the values appearing in

$$\bigcup_{1 \leq j \leq h} \text{BackgroundR}_j \cup \bigcup_{1 \leq j \leq h} \text{IntBackgroundR}_j$$

where $h = \text{height}(A)$ and the constants.

The auxiliary predicates Eval_i^{psi}, IntEval_i^{ψ} and IniEval^{ψ} represent tuples satisfying a positive formula ψ on different relations.

The program $\mathcal{P}_A$ consists of several rules for every transition of A. Recall that as A is progressive, for each transition $d = (s, \psi^- \wedge \psi^+, s')$ of A there is a unique access method AcM that can appear in predicates $\text{IsBind}_{\text{AcM}}$ in $\psi^- \wedge \psi^+$.

We now define the rules for the predicates ViewR_i. For each transition $d = (s, \psi^- \wedge \psi^+, s')$ remaining within component C_i of A, for the unique access method AcM that can appear in predicates $\text{IsBind}_{\text{AcM}}$ in $\psi^- \wedge \psi^+$, the rule:

$$\begin{aligned} \text{ViewR}_i(\pi_{\text{AcM}}(\bar{x}; \bar{v})) \ :- \ & \text{ReachC}_i, \\ & \text{Eval}_{i,\text{AcM}}^{\psi^+}(\bar{x}), \\ & \text{BackgroundR}_i(\pi_{\text{AcM}}(\bar{x}; \bar{v})) \end{aligned}$$

where $\bar{x}$ is a sequence of pairwise disjoint variables and $\pi_{\text{AcM}}(\bar{x}; \bar{v})$ is the permutation of $\bar{x}, \bar{v}$ that preserves the order among $\bar{x}$ and $\bar{v}$ (that is if $\bar{x}(i)$ occurs to the left of $\bar{x}(j)$ in $\pi_{\text{AcM}}(\bar{x}; \bar{v})$ then $i < j$ and likewise for $\bar{v}$) and such that the variables of $\bar{x}$ occur exactly on the positions that are bound by AcM.

For $d = (s, \psi^- \wedge \psi^+, s')$ a transition from C_{i-1} to C_i and $i > 1$, we have the rule:

$$\begin{aligned} \text{ViewR}_i(\pi_{\text{AcM}}(\bar{x}; \bar{v})) \ :- \ & \text{ReachC}_i, \\ & \text{IntEval}_i^{\psi^+}(\bar{x}), \\ & \text{IntBackgroundR}_{i-1}(\pi_{\text{AcM}}(\bar{x}; \bar{v})) \end{aligned}$$

We also have the rule

$$\text{ViewR}_1(\bar{x}) \ :- \ \text{IntBackgroundR}_0(\bar{x}), \text{ReachC}_1$$

where $\bar{x}$ is a sequence of pairwise distinct variables of length arity(R).

For all $1 \leq i \leq h$ we have the following rules for the predicates Val_1^i and $\text{Val}_k^{\leq i}$:

$$\begin{aligned} \text{Val}_1^i(x) \ &:- \ \text{BackgroundR}_i(\bar{y}, x, \bar{z}) \\ \text{Val}_1^i(x) \ &:- \ \text{IntBackgroundR}_{i-1}(\bar{y}, x, \bar{z}) \end{aligned}$$

Here $\bar{y}$ and $\bar{z}$ are two sequences of variables such that their concatenation $\bar{y}, \bar{z}$ contains no variable twice and $|\bar{y}, x, \bar{z}|$ is the arity of R. For each $k \leq m$ where m is the maximal arity of a relation in Sch, the rule

$$\text{Val}_k^i(\bar{x}) \ :- \ \text{Val}_1^i(x_1), \ldots, \text{Val}_1^i(x_k)$$

where $\bar{x} = x_1, \ldots, x_k$ are pairwise distinct variables. For all $1 \leq i \leq h$ and $1 \leq j \leq i$ and for all $c \in C$ the rules:

$$\begin{aligned} \text{Val}_k^{\leq i}(\bar{x}) \ &:- \ \text{Val}_k^j(\bar{x}) \\ \text{Val}_1^{\leq i}(c) \ &:- \end{aligned}$$

For the predicate ReachC_i we have the rule

$$\text{ReachC}_1 \ :- \ \text{IniEval}^{\Upsilon_{\text{pre}}(s_0)}$$

and for each $1 < i \leq \text{height}(A)$ the rule

$$\text{ReachC}_i \ :- \ \text{IntEval}^{\theta_i^+}(\bar{c}_{i-1}), \text{ReachC}_{i-1}$$

where $\theta = \theta^+ \wedge \theta^-$ is the formula on the transition that connects C_{i-1} to C_i.

We have for all $1 \leq i \leq h$ the following rule for Equal_i:

$$\text{Equal}_i(x, x) \ :- \ \text{Val}_1^{\leq i}(x)$$

And for each each constant $c \in C$ the following fact added:

$$\text{Equal}_i(c, c) \ :-$$

The rules for the predicates of the form IniEval^{ψ} are:

- For ψ of the form $R_{\text{pre}}(\bar{y})$ we define

$$\text{IniEval}^{\psi}(\bar{y}) \ :- \ \text{IntBackgroundR}_0(\bar{y})$$

- If $\psi(\bar{y})$ is of the form $t = t'$ where t and t' are both either a variable or a constant in C and $\bar{y}$ are the variables that occur free in ψ, then

$$\text{IniEval}^{\psi}(\bar{y}) \ :- \ \text{Equal}_1(t, t')$$

- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \wedge \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\begin{aligned} \text{IniEval}^{\psi}(\bar{y}) \ :- \ & \text{IniEval}^{\psi_1}(\bar{y}_1), \\ & \text{IniEval}^{\psi_2}(\bar{y}_2). \end{aligned}$$

- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \vee \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\begin{aligned} \text{IniEval}^{\psi}(\bar{y}) \ &:- \ \text{IniEval}^{\psi_1}(\bar{y}_1), \text{Val}_{|\bar{y}|}^1(\bar{y}) \\ \text{IniEval}^{\psi}(\bar{y}) \ &:- \ \text{IniEval}^{\psi_2}(\bar{y}_2), \text{Val}_{|\bar{y}|}^1(\bar{y}) \end{aligned}$$

- If $\psi(\bar{y})$ is of the form $\exists y \,.\, \psi_1(y, \bar{y})$ and $y, \bar{y}$ are exactly the free variables of ψ_1 then

$$\text{IniEval}^{\psi}(\bar{y}) \ :- \ \text{IniEval}^{\psi_1}(y, \bar{y})$$

The rules for the predicates Eval^{ψ} are:

- If $\psi(\bar{y})$ is of the form $R_{\text{pre}}(\bar{t})$, where $\bar{t}$ is a sequence of variables and constants and $\bar{y}$ are exactly the variables in $\bar{t}$, then

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i}_{b}(\bar{x}), \text{ViewR}_1(\bar{t})$$
$$\vdots$$
$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i}_{b}(\bar{x}), \text{ViewR}_i(\bar{t})$$

where b is the number of bound variables of AcM and $\bar{x}$ is a sequence of b pairwise distinct variables that are disjoint from $\bar{y}$.
- If $\psi(\bar{y})$ is of the form $R_{\text{post}}(\bar{t})$, where $\bar{t}$ is a sequence of variables and constants and $\bar{y}$ are exactly the variables in $\bar{t}$, then we add

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i}_{b}(\bar{x}), \text{ViewR}_1(\bar{t})$$
$$\vdots$$
$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i}_{b}(\bar{x}), \text{ViewR}_i(\bar{t})$$

where b is the number of bound variables of AcM and $\bar{x}$ is a sequence of b pairwise distinct variables that are disjoint from $\bar{y}$. In addition, if the relation R is used in the access AcM, we add the rule

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{y}',\bar{y}) \;:-\; \text{BackgroundR}_i(\bar{y}) \qquad (*)$$

where $\bar{y}'$ is the subsequence of $\bar{y}$ that contains exactly those positions of $\bar{y}$ that are bound in the access of d.
- For ψ of the form $\text{IsBind}_{\text{AcM}}(\bar{t})$ where $\bar{t}$ is a sequence of variables and constants, we add the rule:

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{t},\bar{t}) \;:-\; \text{Val}^{i}_{1}(t_{i_1}), \ldots, \text{Val}^{i}_{1}(t_{i_m})$$

where $i_1, \ldots, i_m$ is the sequence of positions of $\bar{t}$ that are associated with variables.
- If $\psi(\bar{y})$ is of the form $t = t'$ where t and t' are both either a variable or a constant in C and $\bar{y}$ are the variables that occur free in ψ, then

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i}_{b}(\bar{x}), \text{Equal}_i(t,t')$$

where b is the number of bound variables of AcM and $\bar{x}$ is a sequence of b pairwise distinct variables that are disjoint from $\bar{y}$.
- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \wedge \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Eval}^{\psi_1}_{i,\text{AcM}}(\bar{x},\bar{y}_1), \text{Eval}^{\psi_2}_{i,\text{AcM}}(\bar{x},\bar{y}_2)$$

where b is the number of bound variables of AcM and $\bar{x}$ is a sequence of b pairwise distinct variables that are disjoint from $\bar{y}$.
- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \vee \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Eval}^{\psi_1}_{i,\text{AcM}}(\bar{x},\bar{y}_1), \text{Val}^{\leq i}_{b}(\bar{y}).$$
$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Eval}^{\psi_2}_{i,\text{AcM}}(\bar{x},\bar{y}_2), \text{Val}^{\leq i}_{b}(\bar{y})$$

where $\bar{x}$ and b are as above (as in the corresponding case of the definition of IniEval^{ψ}, we need to filter by $\text{Val}^{\leq i}_{b}$ in order for the rule to be safe).
- If $\psi(\bar{y})$ is of the form $\exists y \,.\, \psi_1(y,\bar{y})$ and $y, \bar{y}$ are exactly the free variables of ψ_1 then

$$\text{Eval}^{\psi}_{i,\text{AcM}}(\bar{x},\bar{y}) \;:-\; \text{Eval}^{\psi_1}_{i,\text{AcM}}(\bar{x},y,\bar{y})$$

where $\bar{x}$ is as above.

The rules for predicates of the form $\text{IntEval}^{\psi}_{i}$ are:
- If ψ is of the form $R_{\text{pre}}(\bar{t})$, where $\bar{t}$ is a sequence of variables and constants, then

$$\text{IntEval}^{\psi}_{i}(\bar{c}_i,\bar{t}) \;:-\; \text{ViewR}_1(\bar{t})$$
$$\vdots$$
$$\text{IntEval}^{\psi}_{i}(\bar{c}_i,\bar{t}) \;:-\; \text{ViewR}_i(\bar{t})$$

where $\bar{c}_i$ is the constant vector associated with the transition from C_{i-1} to C_i.
- If ψ is of the form $R_{\text{post}}(\bar{t})$, where $\bar{t}$ is a sequence of variables and constants, then we add all rules from the previous case, and also

$$\text{IntEval}^{\psi}_{i}(\bar{c}_i,\bar{t}) \;:-\; \text{ViewR}_1(\bar{t})$$
$$\vdots$$
$$\text{IntEval}^{\psi}_{i}(\bar{c}_i,\bar{t}) \;:-\; \text{ViewR}_i(\bar{t})$$

where $\bar{c}_i$ is the constant vector associated with the transition from C_{i-1} to C_i. For the relation R used in the access AcM of the transition from C_i to C_{i+1}, we add the rule

$$\text{IntEval}^{\psi}_{i}(\bar{c}_i,\bar{y}) \;:-\; \text{IntBackgroundR}_i(\bar{y})$$

- For ψ of the form $\text{IsBind}_{\text{AcM}}(\bar{t})$ where $\bar{t}$ is a sequence of variables and constants, we add the rule:

$$\text{IntEval}^{\psi}_{i}(\bar{t},\bar{t}) \;:-\; \text{Val}^{i+1}_{1}(t_{j_1}), \ldots, \text{Val}^{i+1}_{1}(t_{j_m})$$

where $j_1, \ldots, j_m$ is the sequence of positions of $\bar{t}$ that are associated with variables.
- If $\psi(\bar{y})$ is of the form $t = t'$ where t and t' are both either a variable or a constant in C, and $\bar{y}$ are the variables that occur in ψ, then

$$\text{IntEval}^{\psi}_{i}(\bar{x},\bar{y}) \;:-\; \text{Val}^{i+1}_{b}(\bar{x}), \text{Equal}_{i+1}(t,t')$$

where b is the number of bound positions of the access used by the transition from C_i to C_{i+1} and $\bar{x}$ is a sequence of b pairwise distinct variables that are disjoint from $\bar{y}$.
- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \wedge \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\text{IntEval}^{\psi}_{i}(\bar{x},\bar{y}) \;:-\; \text{IntEval}^{\psi_1}_{i}(\bar{x},\bar{y}_1), \text{IntEval}^{\psi_1}_{i}(\bar{x},\bar{y}_2)$$

where $\bar{x}$ is as above.
- If $\psi(\bar{y})$ is of the form $\psi_1(\bar{y}_1) \vee \psi_2(\bar{y}_2)$, where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are exactly the free variables of ψ, ψ_1 and ψ_2 respectively, then

$$\text{IntEval}^{\psi}_{i}(\bar{x},\bar{y}) \;:-\; \text{IntEval}^{\psi_1}_{i}(\bar{x},\bar{y}), \text{Val}^{\leq i+1}_{b}(\bar{y})$$
$$\text{IntEval}^{\psi}_{i}(\bar{x},\bar{y}) \;:-\; \text{IntEval}^{\psi_2}_{i}(\bar{x},\bar{y}), \text{Val}^{\leq i+1}_{b}(\bar{y})$$

where $\bar{x}$ and b are as above.

- If $\psi(\bar{y})$ is of the form $\exists y\ \psi_1(y, \bar{y})$ and $y, \bar{y}$ are exactly the free variables of ψ_1, then

$$\mathrm{IntEval}_i^{\psi}(\bar{x}, \bar{y}) \ :-\ \mathrm{IntEval}_i^{\psi_1}(\bar{x}, y, \bar{y})$$

where $\bar{x}$ is as above.

We now give the construction of the positive query $\mathcal{P}'_A$, which is the disjunction of $\mathcal{P}''_A$ and $\mathcal{P}'''_A$. Φ_i^- is the smallest set that contains the following formulas:

- Φ_i^- contains each negative sentence of $\Upsilon_{\mathrm{pre}}(s_0)$, where we replace each atom $R_{\mathrm{pre}}(\bar{x})$ by $\mathrm{IntBackgroundR}_0(\bar{x})$.
- for every transition $(s, \varphi^- \wedge \varphi^+, s')$ of A, if s and s' are in the same component C_i, then Φ_i^- contains the formula $\tilde{\varphi}^-$ that is obtained from φ^- by replacing each atom $R_{\mathrm{pre}}(\bar{x})$ or $R_{\mathrm{post}}(\bar{x})$ by

$$\bigvee_{j \leq i} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-1} \mathrm{IntBackgroundR}_j(\bar{x});$$

- for every transition $(s, \varphi^- \wedge \varphi^+, s')$ of A, if s is in C_{i-1} and s' is in C_i, then Φ_i^- contains the formula $\tilde{\varphi}^-$ that is obtained from φ^- by replacing each atom $R_{\mathrm{pre}}(\bar{x})$ by

$$\bigvee_{j \leq i-1} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-2} \mathrm{IntBackgroundR}_j(\bar{x})$$

and each atom $R_{\mathrm{post}}(\bar{x})$ by

$$\bigvee_{j \leq i-1} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-1} \mathrm{IntBackgroundR}_j(\bar{x})$$

We define $\mathcal{P}''_A$ to be $\mathrm{dual}(\bigwedge_{i \leq h} \tilde{\Phi}_i^-)$ where $h = \mathrm{height}(A)$ and $\tilde{\Phi}_i^-$ is the conjunction of all formulas in Φ_i^-, and for a propositional formula ρ, $\mathrm{dual}(\rho)$ is its DeMorgan dual. Note that $\mathcal{P}'_A$ is a positive formula as the DeMorgan dual of a formula is equivalent to its negation.

$\mathcal{P}'''_A$ is a set of "sanity rules" on $\mathrm{IntBackgroundR}_i$. We wish to insure that for every i only one $\mathrm{IntBackgroundR}_i$ is nonempty, and it corresponds to the R associated with the transition from C_{i-1} to C_i. This is enforced by disjuncts of the form $\exists \vec{x}\ \mathrm{IntBackgroundR}_i(\vec{x})$ whenever R is not the relation of the access associated with the transition.

A.6 Correctness of the reduction to Datalog containment

We now prove that the Datalog program $\mathcal{P}_A$ and the positive query $\mathcal{P}'_A$ constructed from a progressive access automaton as described above have the properties required for a reduction. That is, we show that

> A has an accepting run iff there is a database D that is a model of $\mathcal{P}_A$ but not of $\mathcal{P}'_A$.

We start with the more involved direction from right to left. Hence let D be a database over the extensional schema of $\mathcal{P}_A$. We assume that D is a model of $\mathcal{P}_A$ but not of $\mathcal{P}'_A$.

We first present a proposition showing that the Datalog program $\mathcal{P}$ can be decomposed into (not necessarily disjoint) subprograms

$$\mathcal{P}_1, \mathrm{Int}\mathcal{P}_1, \mathcal{P}_2, \mathrm{Int}\mathcal{P}_2, \ldots, \mathrm{Int}\mathcal{P}_{h-1}, \mathcal{P}_h$$

for $h = \mathrm{height}(A)$, that can be evaluated "in sequence". First we let $\mathcal{P}'$ contain all rules of $\mathcal{P}_A$ that do not define predicates of the form ViewR_i or ReachC_i for some i.

- $\mathcal{P}_1$ contains all rules in $\mathcal{P}'$, the rules of $\mathcal{P}_A$ that compute the intensional predicate ViewR_1, but only those that correspond to transitions in the strongly connected component C_1, and the rule for the predicate ReachC_1.
- For $1 < i \leq h$, $\mathcal{P}_i$ contains all rules of $\mathcal{P}'$ and all rules of $\mathcal{P}_A$ that compute the intensional predicate ViewR_i, but only those that correspond to transitions in the strongly connected component C_i.
- For $1 \leq i < h$, $\mathrm{Int}\mathcal{P}_i$ contains all rules of $\mathcal{P}'$, the rule defining ViewR_{i+1} that has been obtained from the transition connecting C_i to C_{i+1}, and the rule defining ReachC_{i+1}.

PROPOSITION A.4. *For every database D over the extensional schema of $\mathcal{P}_A$*

$$\mathcal{P}_A(D) = \mathcal{P}_h(\mathrm{Int}\mathcal{P}_{h-1} \ldots \mathrm{Int}\mathcal{P}_1(\mathcal{P}_1(D)) \ldots)$$

PROOF OF PROPOSITION A.4. It is obvious that the right hand side is included in the left hand side. The other direction follows from the fact that if head $:-$ body is a rule in $\mathcal{P}_i$ then body contains only atoms that appear as heads of rules in $\mathcal{P}_1 \cup \mathrm{Int}\mathcal{P}_1 \ldots \cup \mathcal{P}_i$ and, similarly, if head $:-$ body is a rule in $\mathrm{Int}\mathcal{P}_i$ then body contains only atoms that appear as heads of rules in $\mathcal{P}_1 \cup \mathrm{Int}\mathcal{P}_1 \ldots \cup \mathrm{Int}\mathcal{P}_i$. $\square$

We now derive a sequence of claims that will assist in proving correctness. The first few are concerned with the relationship between the formulas that appear in the automaton A and predicates in the Datalog program $\mathcal{P}_A$. The first claim is that our translation from positive first order formulas into the predicate $\mathrm{IniEval}^{\psi}$ is correct.

CLAIM 3. *Let A be a progressive A-automaton and let $\psi(\bar{x})$ be a subformula of the pure pre formula $\Upsilon_{\mathrm{pre}}(s_0)$. Also, let $I_{\mathcal{P}}$ be an instance for the signature of the Datalog program $\mathcal{P}_A$ and let I_A be the instance over the schema of the automaton A where each relation R is interpreted by $I_{\mathcal{P}}(\mathrm{IntBackgroundR}_0)$. Then for any AcM, any $\bar{b}$, and each valuation ν*

> $\mathrm{IniEval}^{\psi}(\bar{y})$ *holds on* $(I_{\mathcal{P}}, \nu)$ *iff* $\psi(\bar{y})$ *holds on* $((I_A, (\mathrm{AcM}, \nu(\bar{y})), I_A), \nu)$.

We omit the straightforward proof.

The relationship between positive first-order formulas and the predicate $\mathrm{Eval}^{\psi}_{i,\mathrm{AcM}}$ is slightly more complicated, and we need to introduce some notation to formalize it. Given a relation R with access AcM, and given a binding $\bar{b}$ for AcM, we denote by $R|_{\mathrm{AcM},\bar{b}}$ the subset of R that contains exactly those tuples of R that coincide with $\bar{b}$ on the positions bound in AcM. Given an instance I over the schema Sch and a relation $\tilde{R}$ in the schema Sch, we define $\mathrm{I}+\tilde{R}$ to be the instance such that $(\mathrm{I}+\tilde{R})(R) = \mathrm{I}(R)$ if $R \neq \tilde{R}$ and $(\mathrm{I}+\tilde{R})(R) = \mathrm{I}(\tilde{R}) \cup \tilde{R}$ if $R = \tilde{R}$.

CLAIM 4. *Let A be a progressive A-automaton and $\psi(\bar{x})$ a $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula in the formula set of A. Also, let $\mathrm{I}_{\mathcal{P}}$ be an instance for the signature of $\mathcal{P}_A$ and let I^i_A be the instance over the schema of A where each R is interpreted by $\bigcup_{j \leq i} \mathrm{I}_{\mathcal{P}}(\mathrm{ViewR}_j)$. If $\tilde{R}$ is a relation in the schema of A that has an access AcM, $\bar{b}$ is a binding for AcM, t is a tuple, and ν is a valuation, then*

> $\mathrm{Eval}^{\psi}_{i,\mathrm{AcM}}(\bar{x}, \bar{y})$ *holds on* $(I_{\mathcal{P}}, \nu)$ *iff* $\psi(\bar{y})$ *holds on* $((\mathrm{I}^i_A, (\mathrm{AcM}, \nu(\bar{x}))), \mathrm{I}^i_A + \mathrm{I}_{\mathcal{P}}(\mathrm{Background}\tilde{\mathrm{R}}_i)|_{\mathrm{AcM},\nu(\bar{x})}), \nu)$

If IntI_A^i *is the instance of* A *where each* R *is interpreted by*

$$\bigcup_{j \leq i} \mathrm{I}_{\mathcal{P}}(\mathrm{ViewR}_j) \cup \mathrm{I}_{\mathcal{P}}(\mathrm{IntBackgroundR}_{i-1})$$

then $\mathrm{IntEval}^{\psi}_{i,\mathrm{AcM}}(\bar{x},\bar{y})$ *holds on* $I_{\mathcal{P}}$ *iff* $\psi(\bar{y})$ *holds on*

$$((\mathrm{IntI}_A^i, (\mathrm{AcM}, \nu(\bar{x})), \mathrm{I}_A^i + \mathrm{I}_{\mathcal{P}}(\mathrm{Background}\tilde{\mathrm{R}}_i)_{\mathrm{AcM},\nu(\bar{x})}), \nu).$$

The claim can be proved using a straightforward induction on the structure of $\psi(\bar{x})$.

The following claims allow us to translate a formula from the signature of the automaton A to a formula from the signature of the datalog program $\mathcal{P}_A$.

Claim 5. *Let* $\mathrm{I}_{\mathcal{P}}$ *be an instance for the signature of the Datalog program and* $k \leq \mathrm{height}(A)$. *Let* I_A *be the instance over the schema of* A *where each* R *is interpreted by:*

$$\bigcup_{j \leq k} \mathrm{I}_{\mathcal{P}}(\mathrm{BackgroundR}_j) \cup \bigcup_{j \leq k-1} \mathrm{I}_{\mathcal{P}}(\mathrm{IntBackgroundR}_j)$$

Let $\bar{c}$ *be a vector of values and* AcM *an access method. Then for every positive query* φ *and every relation symbol* R, *if* φ *holds on* $(\mathrm{I}_A, (\mathrm{AcM}, \bar{c}), \mathrm{I}_A)$ *then* $\tilde{\varphi}_k$ *holds on* $\mathrm{I}_{\mathcal{P}}$, *where* $\tilde{\varphi}_k$ *is obtained from* φ *by replacing each atom* $R_{\mathrm{pre}}(\bar{x})$ *or* $R_{\mathrm{post}}(\bar{x})$ *by*

$$\bigvee_{j \leq k} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq k-1} \mathrm{IntBackgroundR}_j(\bar{x}).$$

Claim 6. *Let* $\mathrm{I}_{\mathcal{P}}$ *be an instance for the signature of the Datalog program. Let* I_A^i *be the instance of the automaton* A *where each* R *is interpreted by:*

$$\bigcup_{j \leq i-1} \mathrm{I}_{\mathcal{P}}(\mathrm{BackgroundR}_j) \cup \bigcup_{j \leq i-2} \mathrm{I}_{\mathcal{P}}(\mathrm{IntBackgroundR}_j)$$

and let I_A^{i+1} *be the instance of the automaton* A *where* R *is interpreted by*

$$\bigcup_{j \leq i-1} \mathrm{I}_{\mathcal{P}}(\mathrm{BackgroundR}_j) \cup \bigcup_{j \leq i-1} \mathrm{I}_{\mathcal{P}}(\mathrm{IntBackgroundR}_j)$$

Let $\bar{c}$ *be the sequence of constants associated with the transition from* C_{i-1} *to* C_i *and* AcM *the method associated with the transition. Then for every positive query* φ, φ *holds on* $(\mathrm{I}_A^i, (\mathrm{AcM}, \bar{c}), \mathrm{I}_A^{i+1})$ *implies* $\tilde{\varphi}$ *holds on* $\mathrm{I}_{\mathcal{P}}$.

We omit the obvious proofs of these claims.

The final set of claims concern the automaton only – they will be concerned with showing that paths within a component of the automata are "realizable".

Claim 7. *Let* s, s' *be states in the same strongly connected component of* A *and let* $r = d_1, \ldots, d_n$, $d_i = (s_i, \varphi_i, s_{i+1})$ *be a sequence of transitions such that* $s = s_1$ *and* $s_{n+1} = s'$. *If there is a transition* $t_0 = (\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}_0')$ *in the LTS associated with the schema of* A *that is a model of* $\Upsilon_{\mathrm{post}}(s)$, *then there is a path* $t_1 \ldots t_n$, $t_i = (\mathrm{I}_0', (\mathrm{AcM}_i, \bar{b}_i), \mathrm{I}_0')$ *on which* A *has a run* r.

Note that the instance of A does not change on the access path $t_1 \ldots t_n$.

Proof. Let s, s', r, t_0 be given as in the claim. The proof is by induction on the length of r.

Base case. Let s_0 and s_1 be two states of A that are connected by a transition $d_1 = (s_0, \varphi_1, s_1)$ in a strongly connected component. By hypothesis, there is a transition $t_0 = (\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}_0')$ that is a model of $\Upsilon_{\mathrm{post}}(s_0)$. As s_0 and s_1 are in the same strongly connected component of A, it follows from Condition 4 of the definition of a progressive automaton that t_0 is also a model of $\Upsilon_{\mathrm{post}}(s_1)$. By the same condition it follows that t_0 is a model of $\tilde{\varphi}_1$ where $\tilde{\varphi}_1$ is the formula $\exists \bar{x} \,.\, \varphi'$ where φ' is obtained from φ_1 by replacing each atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ by $\bar{t} = \bar{x}$ and by replacing each predicate R_{pre} by R_{post}.

We need to show that φ_1 is satisfiable by a transition of the form $(\mathrm{I}_0', (\mathrm{AcM}', \bar{b}'), \mathrm{I}_0')$ for some AcM' and $\bar{b}'$. To do this, we consider a formula $\hat{\varphi}$ that is "intermediate" between $\tilde{\varphi}_1$ and φ_1: let $\hat{\varphi}$ be the formula $\exists \bar{x}\ \varphi'$ where φ' is obtained from φ_1 by replacing each atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$ by $\bar{t} = \bar{x}$. Hence $\tilde{\varphi}_1$ is the formula obtained from $\hat{\varphi}$ by replacing each predicate R_{pre} by R_{post}.

As $\tilde{\varphi}_1$ is an existential formula, there must be some sequence $\bar{b}$ of values that witness that t_0 is a model of $\tilde{\varphi}_1$. Consider the transition $t = (\mathrm{I}_0', (\mathrm{AcM}, \bar{b}), \mathrm{I}_0')$ where AcM is the (unique) access method used in d_1. We show by induction on the structure of φ' that it is satisfied by t. The only interesting case is when φ' is of the form $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{t})$. In this case φ' must be $\bar{x} = \bar{t}$. By definition, $\bar{t}$ must be unifiable with $\bar{d}$. It follows that t is a model of $\hat{\varphi}$.

Finally, observe that both instances in t are the same. Hence if t is a model of $\hat{\varphi}$ then t is also a model of φ_1. It follows that A has a run d_1 on t.

Induction case. Assume the claim for all states that are connected by a sequence of transitions of length $j < n$. Now consider two states s and s' and a sequence $r = d_1, \ldots, d_{j+1}$ of transitions $d_i = (s_i, \varphi_i, s_{i+1})$ such that $s = s_1$ and $s_{j+1} = s'$. By hypothesis $t_0 = (\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}_0')$ is a model of $\Upsilon_{\mathrm{post}}(s)$. By induction hypothesis, A has a run $d_1, \ldots, d_j$ on a sequence of transitions $t_1 \ldots t_j$ as in the statement. Hence t_j is a model of φ_j. As in the base case, it follows from Condition 4 of the definition of a progressive automaton that t_j is a model of $\tilde{\varphi}_{j+1}$.

The rest of the proof of the claim is as in the base case: there must be some sequence $\bar{b}$ of values that witness that t_j is a model of $\tilde{\varphi}_{j+1}$. Then, one can show, using the same arguments as in the base case, that $t = (\mathrm{I}_0', (\mathrm{AcM}, \bar{b}), \mathrm{I}_0')$ is a model of $\tilde{\varphi}_{j+1}$, where AcM is the (unique) access method used in d_j. It also follows, as in the base case, that t is also a model of φ_j. Therefore A has a run $d_1, \ldots, d_{j+1}$ on $t_1, \ldots, t_j, t$ which completes the proof of the induction step. □

The next claim is very similar to the previous one. The difference is that it is concerned with the formula $\Upsilon_{\mathrm{pre}}(s_0)$ instead of the formula $\Upsilon_{\mathrm{post}}(s)$.

Claim 8. *Let* s, s' *be states in the same strongly connected component of* A *and let* $r = d_1, \ldots, d_n$, $d_i = (s_i, \varphi_i, s_{i+1})$ *be a sequence of transitions such that* $s = s_1$ *and* $s_{n+1} = s'$. *If there is a transition* $t_0 = (\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}_0')$ *in the LTS associated with the schema of* A *that is a model of* $\Upsilon_{\mathrm{pre}}(s_0)$, *then there is path* $t_1 \ldots t_n$, $t_i = (\mathrm{I}_0, (\mathrm{AcM}_i, \bar{b}_i), \mathrm{I}_0)$ *on which* A *has run* r.

Note that the path $t_1 \ldots t_n$ is over the instance I_0, whereas the path in Claim 7 is over I_0'.

PROOF. Assume that there is a transition t_0 of the form $(\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}'_0)$ in the LTS associated with the schema of A that is a model of $\Upsilon_{\mathrm{pre}}(s_0)$. As Υ_{pre} is a pure pre-formula, it is also a model of $(\mathrm{I}_0, (\mathrm{AcM}_0, \bar{b}_0), \mathrm{I}_0)$. Then the claim follows from Claim 7. $\square$

We are now ready to show the main correctness lemma, which implies that if ReachC_i is true in $\mathcal{P}_A$ then A has a run that ends in C_i.

LEMMA A.5. *1. If* $\mathrm{I}_i = \mathcal{P}_i(\mathrm{Int}\mathcal{P}_{i-1} \dots \mathrm{Int}\mathcal{P}_1(\mathcal{P}_1(D)) \dots)$, *for* $1 \le i \le \mathrm{height}(A)$ *and* ReachC_i *holds on* I_i, *then there is an access path on which A has a run that ends in a state in the strongly connected component C_i such that each relation R of the instance of A after the run is interpreted by* $\bigcup_{j \le i} \mathrm{I}_i(\mathrm{ViewR}_j)$.

2. Similarly, if $\mathrm{IntI}_i = \mathrm{Int}\mathcal{P}_i(\mathcal{P}_i \dots \mathrm{Int}\mathcal{P}_1(\mathcal{P}_1(D)) \dots)$ *for* $1 \le i \le \mathrm{height}(A)$ *and* ReachC_{i+1} *holds on* IntI_i, *then there is an access path on which A has a run that ends in a state in the strongly connected component C_{i+1} such that each relation R of the instance of A after the run is interpreted by* $\bigcup_{j \le i+1} \mathrm{IntI}_i(\mathrm{ViewR}_j)$.

PROOF. We prove both statements by mutual induction.

Base Case, Statement 1.

We show the statement for $i = 1$. Assume that ReachC_1 holds on $\mathcal{P}_1(D)$.

For each relation symbol R in the schema of A, let $T_R = \{R(t_1), \dots, R(t_{m(R)})\}$ be the set of atoms, that contains one atom $R(t)$ for each tuple t in $\mathcal{P}_1(D)(\mathrm{ViewR}_1)$. We define $T = \bigcup_R T_R$ and we denote the elements of T by $\{a_1, \dots, a_n\}$. For some subset S of T, we denote by $\mathcal{P}_1(D)|_S$ the subinstance of $\mathcal{P}_1(D)$ in which the relation R contains exactly the tuples in S.

We show that for each $i \le n$, there is an access path $p_1, \dots, p_i$ on which A has a run $r_1, \dots, r_i$ such that each relation R that A stores after the run has the property that

$$(\mathcal{P}_1(D)|_{\{a_1,\dots,a_i\}})(\mathrm{ViewR}_1) \subseteq R \subseteq \mathcal{P}_1(D)(\mathrm{ViewR}_1)$$

This statement implies the base case of Statement 1. The proof of this statement is itself by induction.

Base case of inner induction. Let a_1 be of the form $\tilde{R}(t_1)$ and let ρ be the rule in $\mathcal{P}_1$ that produces $\mathrm{View}\tilde{\mathrm{R}}_1(t_1)$. Note that all rules in $\mathcal{P}_1$ that produce tuples in $\mathrm{View}\tilde{\mathrm{R}}_1$ are of the form

$$\begin{aligned}\mathrm{View}\tilde{\mathrm{R}}_1(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v})) \;:-\;\; & \mathrm{ReachC}_1,\\ & \mathrm{Eval}^{\psi^+}_{1,\mathrm{A\tilde{c}M}}(\bar{x}),\\ & \mathrm{Background}\tilde{\mathrm{R}}_1(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v}))\end{aligned}$$

or of the form

$$\mathrm{View}\tilde{\mathrm{R}}_1(\bar{x}) \;:-\; \mathrm{IntBackgroundR}_0(\bar{x}), \mathrm{ReachC}_1$$

Hence, ρ must be of one of these forms. Let $d = (s, \varphi, s')$ be the transition of A, that ρ is obtained from.

We will show that there is an access path on which A has a run that starts in the initial state s_0 of A and ends in s. Such a path exists due to the following argument: Recall that we assumed that ReachC_1 holds on $\mathcal{P}_1(D)$. Hence we know from the definition of $\mathcal{P}_A$ that $\mathrm{IniEval}^{\Upsilon_{\mathrm{pre}}(s_0)}$ also holds on $\mathcal{P}_1(D)$. Let I_0 be the instance over the schema of A where each relation R is interpreted by $\mathcal{P}_1(D)(\mathrm{IntBackgroundR}_0)$. Then by Claim 3 we have that $(\mathrm{I}_0, (\mathrm{AcM}, \bar{b}), \mathrm{I}_0)$ is a model of $\Upsilon_{\mathrm{pre}}(s_0)$ for any AcM and $\bar{b}$. Hence by Claim 8 there is path p on which A has run r that starts in s_0 and ends in the first state s of d. In addition, the instance at the end of this path is I_0.

If $a_1 = \tilde{R}(t_1)$ was produced by a rule ρ of the second form, then t_1 must be in $\mathcal{P}_1(D)(\mathrm{IntBackground}\tilde{\mathrm{R}}_0)$. In this case the inner induction is trivial, because t_1 is already in the initial instance I_0 of A. Hence, in the following, we assume that a_1 was produced by a rule ρ of the first form.

We will now extend the access path p by a transition p' such that pp' is an access path on which A has a run rd. We will define the new transition p' to have the access method $\mathrm{A\tilde{c}M}$ used in the rule ρ and to have the binding $\bar{b}$ that is the restriction of t_1 to the positions bound in AcM. We define a relation $\tilde{\mathrm{R}}\mathrm{el}$ with the relation symbol $\tilde{R}$ as follows. Let $\{t_1, \dots, t_k\}$ be all tuples in atoms of $T_{\tilde{R}} = \{\tilde{R}(t_1), \dots, \tilde{R}(t_n)\}$. $\tilde{\mathrm{R}}\mathrm{el}$ is the maximal subset of $\{t_1, \dots, t_k\}$ that is of the form $\tilde{R}|_{\mathrm{A\tilde{c}M},\bar{b}}$ (recall that $R|_{\mathrm{AcM},\bar{b}}$ is the subset of R that contains exactly those tuples of R that coincide with $\bar{b}$ on the positions bound in AcM). We define

$$\mathrm{I} := \mathrm{I}_0 \qquad\qquad \mathrm{I}' := \mathrm{I}_0 + \tilde{\mathrm{R}}\mathrm{el}\,.$$

where $\mathrm{I}_0 + \tilde{\mathrm{R}}\mathrm{el}$ is defined above Claim 4. We can now define $p' = (\mathrm{I}, (\mathrm{A\tilde{c}M}, \tilde{b}), \mathrm{I}')$.

To show that pp' is an access path on which A has a run rd we need to verify that the relational structure associated with p' is a model of the formula ψ on d. Recall that ψ is of the form $\psi^- \wedge \psi^+$ where ψ^- is a positive boolean combination of negated $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences that cannot mention the predicate IsBind, while ψ^+ is a $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentence.

We start with verifying that ψ^- is satisfied. Recall that by hypothesis, $\mathcal{P}'_A$ does not hold on D, and that $\mathcal{P}'_A = \mathrm{dual}(\bigwedge_{i \le h} \tilde{\Phi}^-_i)$ where $\tilde{\Phi}^-_i$ contains the formula $\tilde{\psi}^-$ that is obtained from ψ^- by replacing each atom $R_{\mathrm{pre}}(\bar{x})$ or $R_{\mathrm{post}}(\bar{x})$ by

$$\bigvee_{j \le i} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \le i-1} \mathrm{IntBackgroundR}_j(\bar{x});$$

Therefore, $\tilde{\psi}^-$ must hold on D. As ψ^- contains only negated $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formulas, it follows from Claim 5 and the construction of $\tilde{\Phi}^-_i$ from $\Upsilon_{\mathrm{post}}(C_i)$ that ψ^- holds on $(\mathrm{I}', (\mathrm{AcM}', \bar{b}), \mathrm{I}')$. As I is a subinstance of I' this implies that φ^- holds on p', as required.

We now show that p' is a model of ψ^+. Recall that the tuple t_1 is in $\mathcal{P}_1(D)$, and it has been produced using rule ρ. By definition of $\mathcal{P}_1$, $\mathrm{Eval}^{\psi^+}_{1,\mathrm{A\tilde{c}M}}(\bar{b})$ must hold in $\mathcal{P}_1(D)$ ($\bar{b}$ is the binding that we defined earlier). Then it follows from Claim 4 that ψ^+ holds on p'.

It is left to show that for each relation R in the instance I'

$$(\mathcal{P}_1(D)|_{\{a_1\}})(\mathrm{ViewR}_1) \subseteq R \subseteq \mathcal{P}_1(D)(\mathrm{ViewR}_1)$$

This follows easily from our definitions as explained now. Recall that

$$\mathrm{I}' = \mathrm{I}_0 + \tilde{\mathrm{R}}\mathrm{el}$$

and that I_0 only contains tuples in $\mathcal{P}_1(D)(\mathrm{IntBackgroundR}_0)$. By definition, $\tilde{\mathrm{R}}\mathrm{el}$ contains t_1. This shows the left inclusion. The right inclusion holds because, by definition of

$\mathcal{P}$, $\mathcal{P}_1(D)(\text{IntBackgroundR}_0) \subseteq \mathcal{P}_1(D)(\text{ViewR}_1)$. This concludes the proof of the base case of the inner induction.

Induction step of inner induction. By induction we assume that for some $1 \leq i \leq n$, there is an access path $p_1, \ldots, p_i$ on which A has a run $r_1, \ldots, r_i$ such that each relation R that A stores after $r_1, \ldots, r_i$ has the property that

$$\begin{aligned}(\mathcal{P}_1(D)|_{\{a_1,\ldots,a_i\}})(\text{ViewR}_1) &\subseteq R \\ &\subseteq \mathcal{P}_1(D)(\text{ViewR}_1)\end{aligned}$$

The proof is quite similar to the proof of the base case.

Let a_{i+1} be of the form $\tilde{R}(t_{i+1})$ and let ρ be the rule in $\mathcal{P}_1$ that produces $\text{View}\tilde{\text{R}}_1(t_{i+1})$. As all rules in $\mathcal{P}_1$ that produce tuples in ViewR_1 are of the form

$$\begin{aligned}\text{View}\tilde{\text{R}}_1(\pi_{\text{A}\tilde{\text{c}}\text{M}}(\bar{x};\bar{v})) \ :-\ & \text{ReachC}_1, \\ & \text{Eval}^{\psi^+}_{1,\text{A}\tilde{\text{c}}\text{M}}(\bar{x}), \\ & \text{BackgroundR}_1(\pi_{\text{A}\tilde{\text{c}}\text{M}}(\bar{x};\bar{v}))\end{aligned}$$

or of the form

$$\text{View}\tilde{\text{R}}_1(\bar{x}) \ :-\ \text{IntBackgroundR}_0(\bar{x}), \text{ReachC}_1$$

ρ must also be of one of these forms. Let $d = (s, \varphi, s')$ be the transition of A, that ρ is obtained from.

Note that the state s_r, that A reaches after the run $r_1, \ldots, r_i$ is not necessarily the state s in d. Hence we show that there is an r and an access path p such that A has a run $r_1, \ldots, r_i, r$ on $p_1, \ldots, p_i, p$ that ends in s. Assume that p_i is of the form (s_i, φ_i, s_i'). As $r_1, \ldots, r_i$ is a run on $p_1, \ldots, p_i$, p_i must be a model of φ_i. By Condition 2 φ_i implies $\Upsilon_{\text{post}}(s_i')$, thus p_i is also a model of $\Upsilon_{\text{post}}(s_i')$. As in addition, s and s_k are in the same strongly connected component, it follows from Claim 7 that r and p exist such that $r_1, \ldots, r_i, r$ is a run on $p_1, \ldots, p_i, p$. In addition, by Claim 7, the instance at the end of $r_1, \ldots, r_i, r$ is the same as the instance of A at the end of $r_1, \ldots, r_i$.

The case where a_1 is produced by a rule of the second form is trivial. Hence we assume that ρ is of the first form. We now show that there is transition p' such that $p_1, \ldots, p_i, p, p'$ is an access path on which A has a run $r_1, \ldots, r_i, r, d$ where d is as defined above. The definition of p' is like in the base case: We let the access method be $\text{A}\tilde{\text{c}}\text{M}$ used in the rule ρ and $\tilde{b}$ be the restriction of t_{i+1} to the positions bound in $\text{A}\tilde{\text{c}}\text{M}$. Let $T_{\tilde{R}} = \{\tilde{R}(t_1), \ldots, \tilde{R}(t_k)\}$ and let $\tilde{\text{R}}\text{el}$ be a relation with symbol $\tilde{R}$ that contains a maximal subset of $\{t_1, \ldots, t_n\}$ that is of the form $R(\text{A}\tilde{\text{c}}\text{M}, \bar{b})$. We let I be the instance of A after $r_1, \ldots, r_i, r$ and $\text{I}' = \text{I} + \tilde{\text{R}}\text{el}$. We define $p' = (\text{I}, (\text{A}\tilde{\text{c}}\text{M}, \tilde{b}), \text{I}')$.

We omit the argument that the relational structure associated with p' is a model of the formula ψ on d, as it is precisely as in the base case of the inner induction. As in the base case this is a simple consequence of the definitions that each relation R that A stores after $r_1, \ldots, r_i, r, d$ has the property that

$$(\mathcal{P}_1(D)|_{\{a_1,\ldots,a_{i+1}\}})(\text{ViewR}_1) \subseteq R \subseteq \mathcal{P}_1(D)(\text{ViewR}_1)$$

Base Case, Statement 2.

We show Statement 2 for $i = 1$. We define $\text{IntI}_1 = \text{Int}\mathcal{P}_1(\mathcal{P}_1(D))$ and assume that ReachC_2 holds on IntI_1. Recall that the rule for ReachC_2 is of the form

$$\text{ReachC}_2 \ :-\ \text{IntEval}^{\theta_1^+}(\bar{c}_1), \text{ReachC}_1$$

Thus ReachC_1 must also be true in IntI_1. Also, as the rule for ReachC_1 is in $\mathcal{P}_1$, ReachC_1 must also be true in $\text{I}_1 = \mathcal{P}_1(D)$. Therefore we can use the induction hypothesis of Statement 1 to conclude that there is an access path p on which A has a run r that ends in a state s_r in the strongly connected component C_1 such that for each relation R in the schema of A,

$$R = \bigcup_{j \leq 2} \mathcal{P}_1(D)(\text{ViewR}_j)$$

As in the proof of the base case of Statement 1, we need to extend this run to a run after which the instance stored by A contains all ViewR_2 tuples that $\text{Int}\mathcal{P}_1$ adds to $\text{Int}\mathcal{P}_1(D)$. Note that there is only one rule ρ in $\text{Int}\mathcal{P}_1$ that has a predicate ViewR_2 in the head. Let $d = (s, \varphi, s')$ be the transition of A from which ρ was obtained. By definition of $\text{Int}\mathcal{P}_1$, d must be the transition of A that connects the strongly connected component C_1 to the strongly connected component C_2. If s_r is the initial state of A, then we use Claim 8 together with the argument from the base case of the inner induction of Statement 1 to show that there is an access path p' and some r' such that rr' is a run of A on pp' that ends in s. If s_r is not the initial state, then we use the argument from the induction step of the inner induction of Claim 1 together with Claim 7 to show that such p' and r' exist. In both cases r' has the property that the instance stored by A does not change on r'.

Recall that the rule ρ of $\text{Int}\mathcal{P}_1$ must be of the form

$$\begin{aligned}\text{ViewR}_2(\pi_{\text{AcM}}(\bar{x};\bar{v})) \ :-\ & \text{ReachC}_2, \\ & \text{IntEval}_2^{\psi^+}(\bar{x}), \\ & \text{IntBackgroundR}_1(\pi_{\text{AcM}}(\bar{x};\bar{v}))\end{aligned}$$

It follows from Proposition A.4 that all tuples that this rule can ever produce can be produced in one step, after evaluating $\mathcal{P}_1(D)$. We exploit this observation by defining an access path on which A can retrieve all $\text{View}\tilde{\text{R}}_2$ tuples that $\text{Int}\mathcal{P}_1$ adds to $\mathcal{P}_1(D)$ in a single access. This is possible as the arguments in ρ that correspond to the binding contain only the constant vector $\bar{c}_2$ associated with the transition from C_1 to C_2.

Formally, we need to define p'' such that $rr'd$ is a run on $pp'p''$. As in the base case of Statement 1, we choose I to be the instance of A after the run rr'. Let $\text{IntI}_1 \setminus \text{I}_1$ be the instance such that $(\text{IntI}_1 \setminus \text{I}_1)(R) = \text{IntI}_1(R) \setminus \text{I}_1(R)$ for all relations R. Let $\tilde{\text{R}}\text{el}$ be a relation instance for the relation symbol $\tilde{R}$, which corresponds to the relation symbol $\text{View}\tilde{\text{R}}_2$ used in ρ. We define $\tilde{\text{R}}\text{el}$ to contain the tuples in $(\text{IntI}_1 \setminus \text{I}_1)(\text{View}\tilde{\text{R}}_2)$. Note that $\tilde{\text{R}}\text{el}$ must be of the form $\tilde{R}|_{\text{A}\tilde{\text{c}}\text{M},\bar{c}_2}$ where $\text{A}\tilde{\text{c}}\text{M}$ is the access method used in ρ. We define I' to be $\text{I} + \tilde{\text{R}}\text{el}$. We can now define $p'' = (\text{I}, (\text{A}\tilde{\text{c}}\text{M}, \bar{c}_2), \text{I}')$, where $\text{A}\tilde{\text{c}}\text{M}$ is the access method used in ρ.

Verifying that the relational structure associated with p'' is a model of the formula ψ on $d = (s, \varphi, s')$ is as in the base case of the inner induction of Statement 1. It is also a simple consequence of the definitions that each relation $(\text{I} + \tilde{\text{R}}\text{el})(R)$ that A stores after $r_1, \ldots, r_i, r, d$ is interpreted by $\bigcup_{j \leq 2} \text{IntI}_1(\text{ViewR}_j)$.

Induction Case, Statement 1.

We assume both statements for i and define

$$\mathrm{IntI}_i = \mathrm{Int}\mathcal{P}_i(\mathcal{P}_i(\ldots\mathcal{P}_1(D)\ldots))$$
$$\mathrm{I}_{i+1} = \mathcal{P}_{i+1}(\mathrm{IntI}_i).$$

We assume that ReachC_{i+1} is true in I_{i+1}. As the rule defining ReachC_{i+1} is in $\mathrm{Int}\mathcal{P}_i$, ReachC_{i+1} must also be true in IntI_i. Thus, by the second statement, there is an access path p on which A has a run r that ends in a state s_r in the strongly connected component C_{i+1} such that for each relation R in the schema of A,

$$R = \bigcup_{j \leq i} \mathrm{IntI}_i(\mathrm{ViewR}_j)$$

We extend p and r in such a way that the instance at the end of the extended run has the property that for each relation R in the schema of A,

$$R = \bigcup_{j \leq i+1} \mathrm{I}_{i+1}(\mathrm{ViewR}_j).$$

In particular, we must find an access path p' and a run r' on A that adds all ViewR_{i+1} tuples in $\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i$ to the instance stored by A.

For each R in the schema of A, let $T_R = \{R(t_1), \ldots, R(t_{m(R)})\}$ be the set of atoms that contains the atom $R(t)$ for each tuple t in $(\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)(\mathrm{ViewR}_{i+1})$. We define $T = \bigcup_R T_R$ and we denote the elements of T by $\{a_1, \ldots, a_n\}$. For some subset S of T, we denote by $\mathcal{P}_{i+1}(D)|_S$ the subinstance of $\mathcal{P}_{i+1}(D)$ in which the relation R contains exactly the tuples in T_R.

As in the base case of Statement 1, we show that for each $i \leq n$, there is an access path $p, p_1, \ldots, p_i$ on which A has a run $r, r_1, \ldots, r_i$ such that each relation R that A stores after $r, r_1, \ldots, r_i$ has the property that

$$((\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)|_{\{a_1,\ldots,a_i\}})(\mathrm{ViewR}_{i+1}) \subseteq R \subseteq (\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)(\mathrm{ViewR}_{i+1})$$

This statement implies the induction case of Statement 1. The proof of this statement is by induction.

Base case of inner induction. Let a_1 be of the form $\tilde{R}(t_1)$ and let ρ be the rule in $\mathcal{P}_{i+1}$ that produces t_1 in $\mathrm{View}\tilde{\mathrm{R}}_{i+1}$. As all rules in $\mathcal{P}_{i+1}$ that produce tuples in $\mathrm{View}\tilde{\mathrm{R}}_{i+1}$ are of the form

$$\mathrm{View}\tilde{\mathrm{R}}_{i+1}(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v})) \ :- \ \mathrm{ReachC}_{i+1}, \mathrm{Eval}^{\psi+}_{i+1,\mathrm{A\tilde{c}M}}(\bar{x}), \mathrm{Background}\tilde{\mathrm{R}}_{i+1}(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v}))$$

ρ must also be of this form. Let $d = (s, \varphi, s')$ be the transition of A, that ρ is obtained from.

We will show that there is an access path on which A has a run that ends in s. We use the argument from the induction step of the inner induction together with Claim 7 to show that there is an access path p_1' and r_1' such that rr_1' is a run on pp_1' that ends in s.

We now extend the access path pp_1' by a transition p_1'' such that $pp_1'p_1''$ is an access path on which A has a run $rr_1'd$ (we will show the base case of the inner induction with witnesses $p_1 = p_1'p_1''$ and $r_1 = r_1'd$). We define the new transition p_1'' as in the base case of Statement 1: We let p_1'' have the access method $\mathrm{A\tilde{c}M}$ used in the rule ρ and the binding $\tilde{b}$ that is the restriction of t_1 to the positions bound in AcM. We define a relation $\tilde{\mathrm{R}}\mathrm{el}$ with the relation symbol $\tilde{R}$ as follows. Let $\{t_1, \ldots, t_k\}$ be all tuples in atoms of $T_{\tilde{R}} = \{\tilde{R}(t_1), \ldots, \tilde{R}(t_n)\}$. $\tilde{\mathrm{R}}\mathrm{el}$ is the maximal subset of $\{t_1, \ldots, t_k\}$ that is of the form $\tilde{R}|_{\mathrm{A\tilde{c}M},\tilde{b}}$ (recall that $R|_{\mathrm{AcM},\bar{b}}$ is the subset of R that contains exactly those tuples of R that coincide with $\bar{b}$ on the positions bound in AcM). We define

$$\mathrm{I} := \mathrm{I}_0 \qquad\qquad \mathrm{I}' := \mathrm{I}_0 + \tilde{\mathrm{R}}\mathrm{el}\,.$$

where $\mathrm{I}_0 + \tilde{\mathrm{R}}\mathrm{el}$ is defined above Claim 4. We define $p_1'' = (\mathrm{I}, (\mathrm{A\tilde{c}M}, \tilde{b}), \mathrm{I}')$ and $p_1 = p_1'p_1''$ and $r_1 = r_1'd$.

We omit the argument that the relational structure associated with p_1'' is a model of the formula φ in the transition d as it is as in the base case of the proof of Statement 1. It is also easy to see that each relation R that A stores after r, r_1 has the property that

$$((\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)|_{\{t_1\}})(\mathrm{ViewR}_1) \ \subseteq \ R \ \subseteq \ (\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)(\mathrm{ViewR}_1)$$

Induction step of inner induction. By induction we assume that for some $1 \leq i \leq n$, there is an access path $p, p_1, \ldots, p_i$ on which A has a run $r, r_1, \ldots, r_i$ such that the relation R that A stores after the run has the property that

$$((\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)|_{\{a_1,\ldots,a_i\}})(\mathrm{ViewR}_{i+1}) \subseteq R \subseteq (\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)(\mathrm{ViewR}_{i+1})$$

The proof is quite similar to the proof of the base case of the inner induction.

Let a_{i+1} be of the form $\tilde{R}(t_{i+1})$ and ρ be the rule in $\mathcal{P}_1$ that produces the tuple t_{i+1} in $\mathrm{View}\tilde{\mathrm{R}}_{i+1}$. As all rules in $\mathcal{P}_1$ that produce tuples in $\mathrm{View}\tilde{\mathrm{R}}_{i+1}$ are of the form

$$\mathrm{View}\tilde{\mathrm{R}}_{i+1}(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v})) \ :- \ \mathrm{ReachC}_{i+1}, \mathrm{Eval}^{\psi+}_{i+1,\mathrm{A\tilde{c}M}}(\bar{x}), \mathrm{BackgroundR}_{i+1}(\pi_{\mathrm{A\tilde{c}M}}(\bar{x};\bar{v}))$$

ρ must also be of this form. Let $d = (s, \varphi, s')$ be the transition of A, that ρ is obtained from.

As previously, the state s_r that A reaches after the run $r, r_1, \ldots, r_i$ is not necessarily the state s in d. Hence we show that there is an r and an access path p such that A has a run $r_1, \ldots, r_i, r$ on $p_1, \ldots, p_i, p$ that ends in s. This argument is as in the induction step of the inner induction of the base case of Statement 1, just that this time we do not have to worry about the case where s_r is the initial state.

The rest of the proof of the inductive case of the inner induction of the induction step for Statement 1 proceeds exactly as in the inner induction of the base case of statement 1: We show that there is a path for A after which each relation R that A stores has the property that

$$((\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)|_{\{a_1,\ldots,a_{i+1}\}})(\mathrm{ViewR}_1) \subseteq R \subseteq (\mathrm{I}_{i+1} \setminus \mathrm{IntI}_i)(\mathrm{ViewR}_1)$$

We omit the details.

Induction Case, Statement 2.

We now prove Statement 2 of Lemma A.5. The proof is very similar to the proof of the base case. We assume Statement 1 for $i+1$ and Statement 2 for i. Also, we define

$$\mathrm{I}_{i+1} = \mathcal{P}_{i+1}(\ldots\mathrm{Int}\mathcal{P}_1(\mathcal{P}_1(D))\ldots)\,\mathrm{IntI}_{i+1} \quad = \mathrm{Int}\mathcal{P}_{i+1}(\mathrm{I}_{i+1})$$

and assume that ReachC_{i+2} holds on IntI_{i+1}. Recall that the rule for ReachC_{i+2} is of the form

$$\mathrm{ReachC}_{i+2} \ :- \ \mathrm{IntEval}^{\theta_i^+}(\bar{c}_{i+1}), \mathrm{ReachC}_{i+1}$$

Thus ReachC_{i+1} must also be true in IntI_{i+1}. Also, the rule for ReachC_{i+1} is in $\mathrm{Int}\mathcal{P}_i$. Hence ReachC_{i+1} must also be true in $\mathrm{I}_{i+1} = \mathcal{P}_{i+1}(\mathrm{Int}\mathcal{P}_i \ldots \mathrm{Int}\mathcal{P}_1(\mathcal{P}_1(D))\ldots)$. Therefore we can use the induction hypothesis of Statement 1 to conclude that there is an access path p on which A has a run that ends in a state in the strongly connected component C_{i+1} such that for each relation R in the schema of A,

$$R = \bigcup_{j \leq i+1} \mathrm{I}_{i+1}(\mathrm{ViewR}_j).$$

where I_{i+1} is the instance $\mathcal{P}_{i+1}(\mathrm{Int}\mathcal{P}_i \ldots \mathrm{Int}\mathcal{P}_{i+1}(\mathcal{P}_1(D))\ldots)$.

As in the proof of the base case of the Statement 1, we need to extend this run to a run after which the instance stored by A contains all ViewR_{i+2} tuples that $\mathrm{Int}\mathcal{P}_{i+1}$ adds to I_{i+1}. Note that there is only one rule ρ in $\mathrm{Int}\mathcal{P}_{i+1}$ that has a predicate ViewR_{i+2} in the head. Let $d = (s, \varphi, s')$ be the transition of A from which ρ was obtained. By definition of $\mathrm{Int}\mathcal{P}_{i+1}$, d must be the transition of A that connects the strongly connected component C_{i+1} to the strongly connected component C_{i+2}. We use the argument from the induction step of the inner induction for Claim 1 together with Claim 7 to show that such p' and r' exist. In both cases r' has the property that the instance stored by A does not change on r'.

We need to define a transition p'' such that $rr'd$ is a run on $pp'p''$. As in the proof of the base case of Statement 2 one can define p'' in such a way that the relational structure associated with p'' is a model of the formula ψ on $d = (s, \varphi, s')$. In addition, p'' can be defined such that each relation R that A stores after the run $rr'd$ is interpreted by $\bigcup_{j \leq i+1} \mathrm{IntI}_i(\mathrm{ViewR}_j)$. We omit the detail as the proof is exactly as in the base case of Statement 2. This concludes the induction step of Statement 2 of Lemma 4.10, and hence the proof of Lemma A.5. □

We now turn to the other direction of the proof of Lemma 4.10. In this direction we show that if A has an accepting run, then there is a database D that is a model of $\mathcal{P}_A$ but not of $\mathcal{P}'_A$.

Hence let p be an access path on which A has a run r. Let I_0 be the initial instance of A on r. For each $1 \leq i \leq \mathrm{height}(A)$ and each strongly connected component C_i of A, let I_i be the database instance produced by the run r immediately before leaving component C_i. Also, let IntI_i be the instance produced by the run r immediately before entering component C_{i+1} (hence $\mathrm{IntI}_i \setminus \mathrm{I}_i$ contains exactly the tuples that were added on the access performed by the transition connecting C_i to C_{i+1}).

We define D as follows: D contains for each $0 < i \leq \mathrm{height}(A)$ and each relation R in the schema of A, the relations

$$\begin{aligned} \mathrm{IntBackgroundR}_0 &= \mathrm{I}_0(R) \\ \mathrm{BackgroundR}_i &= \mathrm{I}_i(R) \setminus \mathrm{IntI}_{i-1}(R) \\ \mathrm{IntBackgroundR}_i &= \mathrm{IntI}_i(R) \setminus \mathrm{I}_i(R) \end{aligned}$$

We first show that D is a model of $\mathcal{P}_A$. Let I_r be the instance produced at the end of the run r. Note that by the definitions of $\mathcal{P}_A$ for any database D we have for each R in the schema of A

$$\begin{aligned} &\mathcal{P}_A(D)(\mathrm{ViewR}_i) \subseteq \\ &\bigcup_{j \leq i} \mathrm{I}_r(\mathrm{BackgroundR}_j) \cup \mathrm{I}_r(\mathrm{IntBackgroundR}_j) \end{aligned}$$

Conversely, one can show that every t in $\mathrm{I}_r(R)$ is either in ViewR_i or in $\mathrm{IntBackgroundR}_i$ in $\mathcal{P}_A(D)$. Hence we have that for all $i \leq \mathrm{height}(A)$,

$$\begin{aligned} &\bigcup_{j \leq i} \mathrm{I}_r(\mathrm{BackgroundR}_j) \cup \mathrm{I}_r(\mathrm{IntBackgroundR}_j) \\ &\subseteq \mathcal{P}_A(D)(\mathrm{ViewR}_i) \end{aligned}$$

This can be shown easily by induction on the length of the run r. As the final state of A is in the strongly connected component C_h, it follows that ViewR_h is not empty. Therefore the the goal predicate ReachC_h, $h = \mathrm{height}(A)$ is produced and hence D is a model of $\mathcal{P}_A$.

We now show that D is not a model of the positive query $\mathcal{P}'_A$, whose definition we recall here. $\mathcal{P}'_A$ is the disjunction of $\mathcal{P}''_A$ and $\mathcal{P}'''_A$. Φ_i^- is the smallest set that contains the following formulas:

- Φ_i^- contains each negative sentence of $\Upsilon_{\mathrm{pre}}(s_0)$, where we replace each atom $R_{\mathrm{pre}}(\bar{x})$ by $\mathrm{IntBackgroundR}_0(\bar{x})$.
- for every transition $(s, \varphi^- \wedge \varphi^+, s')$ of A, if s and s' are in the same component C_i, then Φ_i^- contains the formula $\tilde{\varphi}^-$ that is obtained from φ^- by replacing each atom $R_{\mathrm{pre}}(\bar{x})$ or $R_{\mathrm{post}}(\bar{x})$ by

$$\bigvee_{j \leq i} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-1} \mathrm{IntBackgroundR}_j(\bar{x});$$

- for every transition $(s, \varphi^- \wedge \varphi^+, s')$ of A, if s is in C_{i-1} and s' is in C_i, then Φ_i^- contains the formula $\tilde{\varphi}^-$ that is obtained from φ^- by replacing each atom $R_{\mathrm{pre}}(\bar{x})$ by

$$\bigvee_{j \leq i-1} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-2} \mathrm{IntBackgroundR}_j(\bar{x})$$

and each atom $R_{\mathrm{post}}(\bar{x})$ by

$$\bigvee_{j \leq i-1} \mathrm{BackgroundR}_j(\bar{x}) \vee \bigvee_{j \leq i-1} \mathrm{IntBackgroundR}_j(\bar{x})$$

We define $\mathcal{P}''_A$ to be $\mathrm{dual}(\bigwedge_{i \leq h} \tilde{\Phi}_i^-)$ where $h = \mathrm{height}(A)$ and $\tilde{\Phi}_i^-$ is the conjunction of all formulas in Φ_i^-, and for a propositional formula ρ, $\mathrm{dual}(\rho)$ is its DeMorgan dual. Note that $\mathcal{P}'_A$ is a positive formula as the DeMorgan dual of a formula is equivalent to its negation.

$\mathcal{P}'''_A$ is a set of "sanity rules" on $\mathrm{IntBackgroundR}_i$. We wish to insure that for every i only one $\mathrm{IntBackgroundR}_i$ is nonempty, and it corresponds to the R associated with the transition from C_{i-1} to C_i. This is enforced by disjuncts of the form $\exists \vec{x} \ \mathrm{IntBackgroundR}_i(\vec{x})$ whenever R is not the relation of the access associated with the transition.

Having recalled the definition, we proceed to the argument. First, we denote by $\Upsilon^-_{\mathrm{pre}}(s_0)$ and $\Upsilon^-_{\mathrm{post}}(s)$, for any state s, the conjunction of negative sentences in $\Upsilon_{\mathrm{pre}}(s_0)$ and $\Upsilon_{\mathrm{post}}(s)$, respectively. In proving this result, we distinguish three cases. We check that D is not a model for the formulas in $\mathcal{P}'_A$ associated with

- the formula $\Upsilon_{\mathrm{pre}}(s_0)$,

- any formula $\Upsilon^{-}_{\text{post}}(s)$, s a state of A
- any formula ψ^{-}, where $(s, \psi^{-} \wedge \psi^{+}, s)$ where there exists an i such that s belongs to C_i and s' belongs to C_{i+1}.

We sketch the first case – the others are similar. We know that the initial transition in the witness access path must satisfy $\Upsilon_{\text{pre}}(s_0)$, and thus in particular must satisfy all negated sentences associated with $\Upsilon_{\text{pre}}(s_0)$. If the negation of the replaced version of a negative sentence of $\Upsilon_{\text{pre}}(s_0)$ was satisfied by D, then using the relationship between D and the run of the automaton, we get a contradiction.

Extension to exact accesses. We now explain the proof that for A-automata non-emptiness is decidable in 2EXPTIME over exact and idempotent paths.

The proof for exact paths will follow the same outline as in the non-exact case, but with three main changes in the construction:

- Again we go through use the conversion to progressive automata from Theorem 4.2, but we will add an extra "repetitivity" requirement on progressive automata, saying that any transition within a component could have happened in a cross-component transition:

  For each transition (s, φ, s') that stays within the same component C_i, there is an "earlier" transition (s_1, φ_1, s_2) where $s_1 \in C_j$ for $j < i$, such that φ_1 implies φ.

  We will state later how a progressive automaton with this additional property can be obtained how the property is used.

  From this, the following strengthening of the 'realizability claim" can be easily shown:

  CLAIM 9. *Let s, s' be states in the same strongly connected component of A, let p be an exact access path and r a run of A on p that begins at the initial state and ends at s. Then there is an exact access path $p; p'$ extending p and a run $r; r'$ extending r such that $r; r'$ ends in s' and the instance does not change on p' and every binding that was used in p' was already used in p.*

  To prove the claim, we have to realize transitions without changing an instance. But given a transition $t = (s, \varphi, s')$ we want to realize, we know that φ is implied by some φ_1, where (s_1, φ_1, s_2) is a transition of an earlier component. Since there is a unique cross-component transition, φ_1 was satisfied by an earlier binding of a transition in p; furthermore the binding must be a constant binding. We can thus use that same binding to realize t.

- There will be a change to the Datalog program formed from the progressive automaton. We will add to the input schema predicates $\text{FirstMade}_{i,\text{AcM}}$ for every component index i and each access method AcM, whose arity is the input arity of AcM. Informally these record which accesses to AcM were made for the first time at i.

  We change the rules of the program to state that for any fact with predicate R occurring at stage i, there must be a compatible binding and access method AcM for R such that the binding satisfies $\text{FirstMade}_{j,\text{AcM}}$ for some $j \leq i$.

- We also add clauses to the positive query prohibiting that a binding occurs in $\text{FirstMade}_{j,\text{AcM}}$ for two distinct j. We also add a clause ensuring that if a tuple is returned for the first time at some stage i, then no compatible binding has been used in a stage below i. This last addition makes use of the new input predicate $\text{FirstMade}_{j,\text{AcM}}$.

The two inductive statements of Lemma A.5 are as before. Claim 9 and the new conditions in the program and in the positive query are applied in the inductive case of their proof.

In the induction, we want to extend a path p to realize a new fact from the Datalog program, associated with a transition d from a state s to a state s' in some component C_i. Using the modification of the positive query and the Datalog program, we know that this fact has a binding b_0 that satisfies $\text{FirstMade}_{k,\text{AcM}_0}$ for some access method AcM_0 $k \leq i$. We also know that the fact is incompatible with every "old" binding – i.e. with every binding that satisfied $\text{FirstMade}_{j,\text{AcM}}$ for any access method AcM and any $j < i$. In particular, we must have b_0 satisfying $\text{FirstMade}_{i,\text{AcM}_0}$. Using Claim 9 we can extend to p to a path $p; p'$ that gets to state s without changing the instance, and which uses only "old" accesses – accesses appearing in p. Using the modification of the Datalog program, we know that the facts corresponding to such accesses must have bindings that satisfy $\text{FirstMade}_{j,\text{AcM}}$ for $j < i$. Thus adding the new fact is compatible with the facts associated with every other access previously done in $p; p'$, so we can extend $p; p'$ with an access to AcM_0 on b, without contradicting exactness. As before, if the access to AcM_0 on b returns several tuples that were not accounted for in prior facts, then the corresponding facts are added in the same inductive stage.

We now explain how we reduce from progressive automaton to a family of progressive automata with the additional "repetitivity property" above. We start with an arbitrary progressive automaton A, and then iterate over all loop-free paths p from a start state to an accept state, forming a new automaton A'_p for each, where A_p will have the required property.

A'_p will be partitioned into connected components of the form A'_t for each transition t in p. The component for t_i will precede that for t_{i+1}. At some stage we have a new transition $t_{i+1} = (s, \varphi, s')$ to process. We let $A_{t_{i+1}}$ be the maximal connected component of s' within the subautomata obtained by restricting A to the states occurring up to t in p. We create a copy of $A_{t_{i+1}}$, $A'_{t_{i+1}}$ in A' as a new component. We add a transition from the copy of s in A'_{t_i} to the copy of s' in $A'_{t_{i+1}}$.

One can verify that this automaton has the required properties.

For idempotent accesses, we perform the same transformations, but for the positive query we only add clauses to prohibiting that a binding occurs in $\text{FirstMade}_{j,\text{AcM}}$ for two distinct j.

A.7 Proof of Proposition 4.11

We recall the statement:

> The containment problem of a Datalog program P in a positive first-order sentence φ is in 2EXPTIME.

We first show the argument in the absence of constants and equality atoms. We adapt the proof of that containment

of a Datalog program P in a union of conjunctive queries can be decided in 2EXPTIME [8]. First, we briefly recall the idea of this proof. Let P be a Datalog program and let $\bigcup_i \theta_i$ be a union of conjunctive queries. The main idea is to associate P with a non deterministic tree automata, A_P, of size exponential in the size of P. In the same way, each θ_i is associated with a non-deterministic tree automata, A_{θ_i}, which is of size exponential in P and θ_i. Theorem 5.11 of [8] states that P is included in $\bigcup_i \theta_i$ iff A_P is included in $\bigcup_i A_{\theta_i}$.

Next, we explain how to use this theorem for the case of positive queries. Let φ be a positive query and $\bigcup_i \theta'_i$ be a union of conjunctive queries equivalent to φ. Without loss of generality, we can assume that the number of formulas θ'_i is exponential in the size of φ and each θ_i is polynomial in the size of φ. For each i, $A_{\theta'_i}$ is the tree automaton associated with θ'_i in Theorem 5.11 of [8]. It is known that for any non-deterministic tree automata A and A', there exists an automaton A'' such that A'' is equivalent to the union of A and A' such that $|A''| \in O(|A| + |A'|)$ [10]. We can deduce that there exists a non-deterministic tree automaton A_φ of size exponential in φ equivalent to $\bigcup_i A_{\theta'_i}$. Moreover the inclusion of two non deterministic tree automata A' and A'' is EXPTIME in their sizes.

So it follows from this and Theorem 5.11 of [8] that the problem of containment of a Datalog program in a positive query is in 2EXPTIME.

To handle constants and equality atoms, we can translate a containment problem of P and φ to another containment problem P' and φ' where there are no constants or equality atoms. This is done by considering an extended signature with unary predicate symbols for each constant symbol, and rewriting P and φ to be disjunctions of constant- and equality- free queries in the larger signature. The disjunction considers the ways in which repeated variables can be realized by a constant c, replacing the repeated variable by distinct variables that satisfy the unary predicate for c. This can be done without a blow-up in the Datalog program P, by introducing an intensional predicate $Eq(x, y)$ that is defined by a disjunction. $Eq(x, y)$ is then be re-used in every other rule. Within φ, a blow-up does occur, so we have thus reduced to checking whether P is contained in $\bigvee_i \varphi_i$, where the disjunction is exponential. By expanding φ_i, we can assume an the right-hand side is an exponential disjunction of conjunctive queries, and then proceed as in the case without constants above.

A.8 Proof of Theorem 4.7

We recall the result:

> Satisfiability of AccLTL$^+$ and emptiness of A-automata are 2EXPTIME-hard

We prove only the statement about A-automata; the same proof technique applies to the logic.

We reduce the containment problem of a Datalog program in a union of conjunctive queries to our problem. This problem is known to be 2EXPTIME-hard [7].

Let Sch be a schema, $P = (q, \mathcal{R})$ a Datalog program with input relations in Sch with head predicate q, and φ be a positive query over Sch. We denote by Sch_{idb} the set of intentional relations used in the rule of P. For each $R \in \mathrm{Sch} \cup \mathrm{Sch}_{idb}$, we have an access method AcM_R on it with all positions as input.

Let $A = (S, S_0, F, \delta)$ be the following automata over the schema $\mathrm{Sch} \cup \mathrm{Sch}_{idb}$:

- $S = \{s_0, s_f\}$, $S_0 = \{s_0\}$, $F = \{s_f\}$,
- For each $R \in \mathrm{Sch}$, there exists a transition d in δ, $d = (s_0, \neg\varphi^{pre} \wedge \neg\varphi^{post} \wedge \exists\vec{x}\, \mathrm{AcM}_R(\vec{x}), s_0)$
- For each rule $r = R(\vec{x}) \ :-\ B_r(\vec{x})$, there exists a transition d in δ, $d = (s_0, \neg\varphi^{pre} \wedge \neg\varphi^{post} \wedge \exists\vec{x} B_r^{pre}(\vec{x}) \wedge \mathrm{AcM}_R(\vec{x}), s_0)$.
- For each rule $r = q() :-\ B_r()$, there exists a transition $(s_0, B_r^{pre}(), s_f) \in \delta$.

Above, for any query Q over Sch, Q^{pre} is the query obtained by changing R to R_{pre}.

Thus the automaton simply checks that φ is never satisfied, and that whenever we have the body of a rule satisfied, we do an access on the head predicate. It then accepts if the goal predicate is ever satisfied. Clearly, if the automaton accepts a path, the final instance in the path cannot satisfy φ, and will have a chain of witnesses for the goal predicate. The automaton does not enforce that the intensional predicates of this instance represent a least fixedpoint (i.e. that they have their appropriate definitions). But if we take the fixedpoint of the resulting configuration, it will still satisfy $\neg\varphi$, since we are changing only intensional predicates.

Conversely, suppose there is an instance I satisfying the Datalog query P along with $\neg\varphi$; we will construct a path that is accepted in the automaton. We start with an access path p_0 that does membership tests for all tuples in I, obtaining true as a result. We then consider the chain of rule instantiations $f_1 \dots f_n$ that witness the truth of P on I. Each f_i can be identified with a grounding of a rule r_i with head predicate $H(\vec{x})$ in P, with $\bar{b}_i$ being the corresponding evaluation of the variables in the body. For each f_i we have an access AC_i using method AcM_H on the restriction of $\bar{b}_i$ to $\vec{x}$, with the response being true. It is easy to check that the access path formed from concatenating p_0 with the accesses AC_i and their responses is accepted by A.

Exact and Idempotent accesses. The above argument was under the standard semantics for access paths. But we note that in the reduction, we used a schema with only one access method per relation, and only boolean accesses. Hence every path is exact for such a schema. Thus we also have hardness when the semantics is restricted to exact accesses and idempotent accesses.

We now give the formal proof that the reduction is correct. Given an instance I for the schema $\mathrm{Sch} \cup \mathrm{Sch}_{idb}$, we let $\mathrm{I} \,|\, \mathrm{Sch}$ (resp. $\mathrm{I} \,|\, \mathrm{Sch}_{idb}$) be the restriction to the relations in Sch (resp. Sch_{idb}).

From automaton non-emptiness to non-containment. We prove that for each access path ρ recognized by A, ending with instance I_n, $\mathrm{I}_{n+1} \,|\, \mathrm{Sch}$ is a witness to non-containment of P in φ. We first show that this instance satisfies the Datalog query P. To do this we show that at any instance I_i on the path, for each intensional predicate R in P, $\mathrm{I}_i(R)$ is a subset of the tuples calculated by P for R on $\mathrm{I}_i \,|\, \mathrm{Sch}$.

Base case. $\rho = \emptyset$, the initial instance is empty, so clearly containment holds.

Induction step. Let ρ be an access path recognized by A of length $i + 1$. By the induction hypothesis, $\mathrm{I}_i(R)$ is included in the value of R computed by P on $\mathrm{I}_i \,|\, \mathrm{Sch}$. We denote by d the last transition associated to the last access in ρ, and divide up into cases.

- the transition was on an access to a predicate $R \in \mathrm{Sch}$. Such accesses are unconstrained by the automaton A, so they may bring a new tuple into extensional relation R. In moving from I_i to I_{i+1} The values of the intensional predicates are unchanged, but by monotonicity of the Datalog program P, the values of the intensional predicates calculated by the program can only increase when moving from $\mathrm{I}_i \,|\, \mathrm{Sch}$ to $\mathrm{I}_{i+1} \,|\, \mathrm{Sch}$. Hence the inductive invariant is preserved.
- d is associated with an access to $R \in \mathrm{Sch}_{idb}$. We consider the case where the matching automaton transition is not the one associated with the rule for the head predicate q. Thus the access may bring a new tuple t into R, and for some rule $r = R(\vec{x}) : -B_r(\vec{x})$ of P t must satisfy $B_r^{pre}(\vec{x})$ in the transition structure, hence t must satisfy $B_r(\vec{x})$ in I_i. By the induction hypothesis, t satisfies the corresponding predicates computed by $P(\mathrm{I}_i \,|\, \mathrm{Sch})$. Thus t satisfies R in $P(\mathrm{I}_i \,|\, \mathrm{Sch})$ and hence by monotonicity satisfies R in $P(\mathrm{I}_{i+1} \,|\, \mathrm{Sch})$.

  The case where the transition does correspond to the head predicate q of P is similar.

Let ρ be a path accepted by A. By construction, the restriction of the final instance I_n to intensional predicates is a subrelation of the intensional predicates calculated by the Datalog program P. In addition, since a final state is reached, the goal predicate must be non-empty. Thus (again by monotonicity), the Datalog query P must be true on the restriction of I_n. On the other hand, the negation of the positive query φ is globally required to hold on all instances I_i in the ρ, hence it holds in the final instance. Thus we have that I_n is a witness of the failure of containment of P in φ.

From non-containment to automaton non-emptiness. Let I be a witness of the failure of containment of P in φ. Let $P_1(\mathrm{I}) \dots P_n(\mathrm{I})$ be successive approximations of the fixpoint P. We prove by induction that for each i, there exists a path ρ whose final instance I_i has its restriction to Sch predicates being I and its restriction to the intensional predicates matching the values calculated in $P_i(\mathrm{I} \,|\, \mathrm{Sch})$.

Base case. We take a path that populates only the predicates of Sch so that they match I.

Induction step. Let ρ be a path recognized by A associated to $\mathrm{I}_i = P^i(\mathrm{I})$, which exists by induction. We show how to extend ρ, abiding by the automaton rules, mimicing each rule in P firing on $\mathrm{I}_i \,|\, \mathrm{Sch}$. We consider a rule r, focusing on the case where r is of the form $R(\vec{x}) : -\, B_r(\vec{x})$ (the case of the head rule is similar). Consider an arbitrary tuple t such that $B_r(t)$ is satisfied in $P^i(\mathrm{I})$. By construction the corresponding transition is possible from I_i by reading the access $(\mathrm{AcM}_R, t, \{t\})$. We thus extend ρ by adding on transitions for each such t. □

A.9 Further Details of the Proof of Theorem 4.12

Recall the result:

> Satisfiability of an $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ formula (over all access paths) is PSPACE-complete. The same holds over exact access path, idempotent access paths, and paths that are both exact and idempotent.

We refer to the proof in the body of the paper, and we explain the modifications needed for idempotent and exact accesses. We first claim that Lemma 4.13 still holds with these conditions. For idempotent accesses, we can use the same process as in the proof of Lemma 4.13: if the original path was idempotent, the diminished path will be as well. For exact access paths, we must ensure that when we shrink the size of instances we do not destroy exactness. We can think of performing the shrinking in two stages, where the first stage involves throwing in query witnesses as in the proof of Lemma 4.13 for general access paths. After doing this, we may have lost exactness: we may have two accesses a_i and a_j in the initial path p (prior to shrinking), where the shrinking maintains a certain tuple t in a_i but throws the same t out of a_j. But we can repair this by just making sure that a tuple is left in a_j if it is left in any other a_i.

Given Lemma 4.13, the rest of the argument proceeds via rewriting as above.

A.10 Proof of Theorem 4.14

Recall the theorem's statement:

> The satisfiability of $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ is Σ_2^P-complete, even when certain accesses are restricted to be idempotent or exact.

We start out with the proof for general accesses.

Hardness The non-containment of positive relational queries under finite types can be reduced to the complement of the satisfiability problem of either language – this problem is known to be Π_2^P-hard.

Upper-Bound. Let an $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ formula φ be given. We make use of the "boundedness lemma", Lemma 4.13, which can be seen also to hold for $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$:

An $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ formula φ is satisfiable iff there exists a path ρ that satisfies the following properties:

- The instances have sizes bounded by a polynomial function in the sizes of φ, and Sch.
- The set of bindings used in ρ has size bounded by a polynomial function in the sizes of φ

We will adapt the technique used in the proof of Theorem 4.12.

Our algorithm will guess first of all a sequence $\bar{\mathrm{I}}$ of instances and a sequence $\bar{\mathrm{AC}}$ of accesses of size polynomial in φ. Let B be the set of bindings used in the sequence $\bar{\mathrm{AC}}$ and let Q be the set of positive subformulas of Φ in which all predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ have been removed (as in the proof of Lemma 4.13).

We denote by $T(\bar{\mathrm{I}}, B)$ the set of transitions of any of the following forms:

- Transitions of form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$ where $\vec{b}$ is in B and compatible with AcM.
- Transitions of form (I_i, A_i, I_{i+1})

For each i, we denote by $T(i)$ the set of transitions of the form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$. For each i, we denote by $t_{i,\rightarrow}$ the transition $(I_i, A(i), I_{i+1})$.

For each transition $t \in T(\bar{\mathrm{I}}, B)$, we let Q_t^+ be the elements of Q which are satisfied by t and we let Q_t^- be the elements of Q that are not satisfied by t. For each query q in Q_t^+, there is a witness vector $v_{q,i}$ for the satisfaction of q in t. By making calls to NP and co-NP subalgorithms, our algorithm

can verify that each of these guesses is correct – e.g. that each query in Q_t^+ really is satisfied in t.

We now translate φ into a propositional LTL_X formula ψ such that ψ is satisfiable iff there is a model of φ that satisfies the regular expression $T(0)^*, t_{0,\rightarrow}, \ldots, t_{n-1,\rightarrow} T(n)^*$. Moreover, the translation from φ to ψ can be obtained in polynomial time. The main difference from the proof of Theorem 4.12 is that we need to express the "sanity axioms" using only the operator X, whereas in the proof of Theorem 4.12, these axioms are encoded using the operator G. However, we notice that the constraints imposed by the formula φ on the path are all restricted to the first $|\varphi|$ accesses. Hence, the "sanity axioms" have to be checked only on the initial $|\varphi|$ accesses, and can thus be rewritten using only the next-time operator X.

Finally, our algorithm checks the satisfiability of the rewritten sentence ψ, which can be done in NP.

We now explain the revision when access methods are required to be exact or idempotent. The conclusion of Lemma 4.13 still applies in this case, and our algorithm begins as before by guessing instances and accesses, and guessing and verifying the positive queries that hold in each instance. One will need to verify as well that the accesses satisfy idempotence or exactness, as required by the underlying access method. Again it suffices to consider paths in

$$T(0)^*, t_{0,\rightarrow}, \ldots, t_{n-1,\rightarrow}, T(n)^*$$

We do this using the same propositional rewriting as above. Note that accesses in $T(i)^*$ always return empty, and are all incompatible with each other and the accesses in $t_{i,\rightarrow}$. Therefore these accesses are always exact. The others accesses have already been verified to have the required properties.

Additional Note. We notice that the conclusion of Lemma 4.13 also holds for $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$. The correctness of the queries can be checked in the same complexity for positive first-order formulas with inequalities. So, the previous arguments are still correct in this context. We can conclude that these results can be extended for formulas with inequalities.

A.11 Proof of Theorem 5.2

Recall the result:

> Satisfiability of binding-positive $\mathrm{AccLTL}(\mathrm{FO}^{\exists+,\neq}_{\mathrm{Acc}})$ queries is undecidable

Again we reduce the problem of implication of functional dependencies (fds) and inclusion dependencies (ids) for relational databases to the problem of the unsatisfiability of a AccLTL^+ with inequalities formula. For simplicity we assume all positions carry the same type. We will also verify the reduction over instances where all relations are nonempty, allowing us to avoid certain corner cases.

Let Γ be a set of inclusion and functional dependencies, and σ be a functional dependency over Sch.

We first give the schema, which extends the relational schema Sch for the dependencies. For each relation R of arity k, we have a relation $\mathrm{Succ}(R)$ of arity $2k$: informally, $\mathrm{Succ}(R)$ will be the successor relation referred to above. There are two relations $\mathrm{Beg}(R)$ (with boolean access $\mathrm{IsBind}_{\mathrm{Beg}(R)}$) and $\mathrm{End}(R)$ (having boolean access $\mathrm{IsBind}_{\mathrm{End}(R)}$) with the same arity as R; these will store the minimal and the maximal tuples for the ordering generated by $\mathrm{Succ}(R)$. In addition there are relations $\mathrm{CheckIncDep}(R)$ with the same arity as R, having boolean accesses $\mathrm{IsBind}_{\mathrm{CheckIncDep}(R)}$. There are used to check the inclusions dependencies for R.

We now explain the formulas that check the various constraints.

First, we verify that each relation $\mathrm{Succ}(R)$ represents a total order on k-tuples. To do this, we first check that positions $1 \ldots k$ and $k+1 \ldots 2k$ are primary keys for $\mathrm{Succ}(R)$. This can be done by adding a conjunct:

$$\mathsf{G}\big(\neg\exists\vec{s}\vec{t}\vec{u}\ \mathrm{Succ}(R)(\vec{s},\vec{t}) \wedge \mathrm{Succ}(R)(\vec{s},\vec{u}) \wedge t_i \neq t_i'\big)$$

Similarly, we can enforce that at any point relation $\mathrm{Beg}(R)$ has only one tuple in it, and that this tuple does not have a predecessor in $\mathrm{Succ}(R)$ – and similarly for $\mathrm{End}(R)$.

The main conjunct of our sentence will assert that we fill the successor relations, and finally move into a "verification phase". We will omit the description of the first phase, which uses a variation of the construction to fill the successor relations in Theorem 3.1. We describe only the subformulas specifying the verification phase. We will focus on the formulas enforcing an inclusion dependency id from relation R to relation S, which are of the form:

$$\begin{aligned}
&\exists\vec{t_1}\ \mathrm{IsBind}_{\mathrm{Beg}(R)}(\vec{t_1}) \wedge \mathsf{X}\ \exists\vec{t_2}\ \mathrm{Beg}(R)_{pre}(\vec{t_2}) \\
&\wedge \mathrm{IsBind}_{\mathrm{CheckIncDep}(id)}(\vec{t_2}) \wedge \\
&\mathsf{X}[(\exists\vec{t}\vec{u}\ \mathrm{Succ}_{pre}(R)(\vec{t},\vec{u}) \wedge \mathrm{CheckIncDep}(id)_{pre}(\vec{t}) \wedge \\
&\mathrm{IsBind}_{\mathrm{CheckIncDep}(id)}(\vec{u}) \\
&\wedge \exists\vec{v}\ \exists\vec{w}(\mathrm{Succ}(S)_{pre}(\vec{v},\vec{w}) \vee \mathrm{Succ}(S)_{pre}(\vec{w},\vec{v})) \wedge \bigwedge_i u_i = v_i) \\
&\mathsf{U}\,(\exists\vec{w}\ \mathrm{End}(R)_{pre}(\vec{w}) \wedge \mathrm{CheckIncDep}(id)(\vec{w}))]
\end{aligned}$$

This describes a sequence s that begins by doing an access to $\mathrm{Beg}(R)$, and then using a tuple that is in $\mathrm{Beg}(R)$, performing an access to $\mathrm{CheckIncDep}(id)$ with the tuple. Assuming that this sequence began in a configuration where $\mathrm{Beg}(R)$ was empty, this would mean that s begins by adding a tuple to $\mathrm{Beg}(R)$ and then accessing $\mathrm{CheckIncDep}(id)$ on the same tuple. The formula states that s will then continue doing accesses to $\mathrm{CheckIncDep}(id)$, using tuples whose predecessor is already in $\mathrm{CheckIncDep}(id)$ and which have a witness for id in the successor relation for S. s will only stop when the last tuple in the order is in $\mathrm{CheckIncDep}(id)$.