Analysis of Schemas with Access Restrictions

Michael Benedikt, Pierre Bourhis, Clemens Ley

Analysis of Schemas With Access Restrictions

MICHAEL BENEDIKT, University of Oxford
and PIERRE BOURHIS, CNRS LIFL University of Lille I and INRIA Lille Nord Europe
and CLEMENS LEY

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.

Categories and Subject Descriptors: H.3.5 [**Information Systems**]: Online Information Services—*Web-based services*; F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic—*Temporal logic*

General Terms: Theory; Verification; Languages

Additional Key Words and Phrases: Access methods, optimization, Hidden Web

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 [Li and Chang 2001; Calì et al. 2009]. 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 predetermined indexed accesses [Ullman 1989]. 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 $\mathrm{Mobile\#}$ table, entering these into the $\mathrm{Address}$ 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 $\mathrm{Mobile\#}$, then this process will not yield Jones' tuple in $\mathrm{Address}$. In general it is known [Li 2003] 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 [Calì et al. 2009; Benedikt et al. 2011] 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 [Benedikt et al. 2011] which proposes limiting the number of accesses to be explored by determining that some accesses are not "relevant" 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. [Benedikt et al. 2011] 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 $\mathrm{Mobile\#}$ 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 $\mathrm{Mobile\#}$, the interface may require a web user to have made at least one access to $\mathrm{Address}$.
— *Dataflow restrictions*; before performing an access to $\mathrm{Mobile\#}$ on a name, the web user must have received that name as a response to a call to $\mathrm{Address}$.

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 lan-

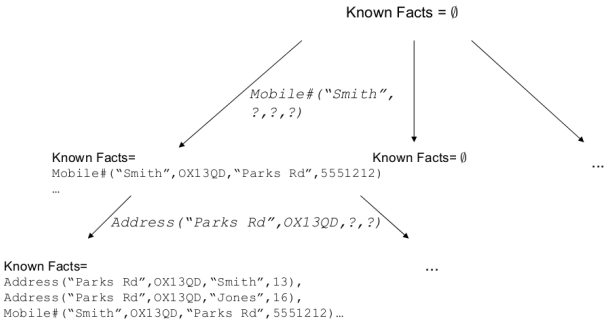

Fig. 1. Tree of possible paths associated with a schema

guage 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* [Emerson 1990]. 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:

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

The relational query prior to the "until" symbol U states that there are no entries in $\mathrm{Mobile}\#_{\mathrm{pre}}$ – the $\mathrm{Mobile}\#$ 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 $\mathrm{Address}$ table prior to the access. Hence this "meta query" returns the set of access paths which have no entries revealed in relation $\mathrm{Mobile}\#$ until an access AC is performed, where AC has method AcM_1 and uses a name that already exists in the $\mathrm{Address}$ 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 denote the logic containing the above sentence by $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, 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 sub-

language 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 queries that are "binding positive", 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 Datalog containment problems, and we in turn show that these Datalog containment problems are decidable. 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, but no dataflow restrictions.
— We determine the impact of 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$. The longest section of the paper, Section 4, deals with the restricted language AccLTL^+. On the way to showing decidability of AccLTL^+, this section defines the related automaton model, explains the connection with certain Datalog containment problems, and provides new decidability results for Datalog containment. Section 6 discusses extensions of AccLTL^+. Section 7 gives conclusions and overviews related work.

An *electronic appendix*, available via the Digital Library, contains additional material filling in details of a few proofs, particularly the main decidability proof of Section 4.

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" [Abiteboul et al. 1995]. A schema Sch includes a set of relations $\{S_1 \ldots S_n\}$, with each S_i associated with a function from $\{1 \ldots n_i\}$, where n_i is the arity of S_i, to Types. We refer to the set $\{1 \ldots n_i\}$ as the *positions* of S_i and the output of the function on j as the *domain* of the j^{th} 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 domains. For two tuples $\bar{t}$ and $\bar{t}'$, we denote by $\bar{t} \cdot \bar{t}'$, the concatenation of the two tuples.

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* $\mathrm{Inp}(\mathrm{AcM})$. Informally, each access method allows one to input a tuple of values for $\mathrm{Inp}(\mathrm{AcM})$ 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. $\mathrm{Mobile\#}(\text{“Jones”}, ?, ?, ?)$ is an access to relation $\mathrm{Mobile\#}$ asking for all phone number information for people named “Jones”. A boolean access is $\mathrm{Mobile\#}(\text{“Jones”}, \text{“OX13QD”}, \text{“Parks Rd”}, 23)?$, 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 r of tuples 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*. Note that we consider a general model where distinct executions of an access can return different responses, thus allowing non-determinism.

A sequence $((\mathrm{AcM}_1, \bar{b}_1), \mathrm{r}_1), \ldots, ((\mathrm{AcM}_n, \bar{b}_n), \mathrm{r}_n)$ of accesses and well-formed responses for some instance I is an *access path for the instance* I. Notice that if we have an access path as above, it will always be the case that each r_i consists of tuples for the relation of AcM_i, and its projection onto the input positions of AcM_i will consist of exactly $\bar{b}_i$. Any sequence with this property will be referred to 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 the *configuration returned by p on I_0*, is the instance where relation S_i contains with all tuples returned by any access to S_i in p. If we take an access path and look at the sequence of configurations, returned by longer and longer prefixes, we will see that the relations grow monotonically, since we only add tuples during accesses. We will generally use the word “configuration” to emphasize that this is a database instance containing information made visible by the access, as opposed to the full information in the hidden instance. In some situations, it is useful to deal with the case where there is some initial instance Conf_0 representing information known in advance (e.g. some well-known facts). Given such a Conf_0 and a path p, we can then talk about the configuration returned by p on Conf_0, which appends to each relation Conf_0 the responses to the accesses in 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 $p = a_1, r_1 \ldots$ 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 $(\mathrm{No}, \mathrm{L}, \mathrm{T})$ 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 we can associate a labelled transition system where the nodes are all the instances 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 the configuration resulting from the access AC and response r is 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) [Emerson 1990]. 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) [Emerson 1990], 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 investigation of a language which allows arbitrary nesting is left for future work.

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 $R \in \mathrm{Sch}$, and also 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 i^{th} 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. When we deal with computational problems in our logics, we will always be restricting the interpretation of the predicates in the way described above. For example the statement in our logic that whenever $R_{\mathrm{post}}(x)$ holds in one transition then $R_{\mathrm{pre}}(x)$ holds in the next transition will be *valid* whenever our logic, when it is clearly not valid over arbitrary interpretations of R_{post} and R_{pre}.

Notice that in the paragraph above we have alluded to a version of FOLTL in which one can only describe sentences about each individual first-order structure, nesting temporal operators on top of them. When such a logic can range over arbitrary sequences of first-order structures, they are known to have very restricted expressiveness, as well as more attractive decidability properties [Hodkinson et al. 2000]. In contrast, we are restricting our sequences to be of a special form, with consecutive structures being related by an access. More importantly, via the formula IsBind we can express some information about the "dynamics" between consecutive structures. As shown above, it is the addition of IsBind and its relationship with the pre- and post- versions of the relations, enforced in our semantics, that will allow us to express many natural properties of access sequences, such as time-invariance. But it is also this feature that will make decidability more problematic.

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

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:

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

In analyzing accesses to data, the paths we are interested in will be the finite ones, We will thus use a semantics for the logic via positions within finite paths. The distinction from the standard semantics of LTL will only be in the interpretation of the next operator, which is interpreted as false at the end of paths. The restriction to finite paths will also allow us to use a simpler kind of automaton to analyze the logic, avoiding the acceptance conditions needed in automata for infinite objects. The semantics of $\mathrm{AccLTL}(L)$ is given by the relation $(p, i) \vDash \varphi$, where $p = t_1 \dots t_n$ 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) $(p, i) \vDash \varphi$ iff $\varphi \in L$ and $M(t_i)$ satisfies φ in the usual sense of first-order logic. (2) $(p, i) \vDash \neg\varphi$ iff $(p, i) \nvDash \varphi$. (3) $(p, i) \vDash \mathsf{X}\varphi$ iff $(p, i + 1) \vDash \varphi$. (4) $(p, i) \vDash \varphi \mathsf{U}\, \psi$ iff there exists $j \geq i$ such that $(p, j) \vDash \psi$ and $\forall i \leq k < j, (p, k) \vDash \varphi$. (5) $(p, i) \vDash \varphi \vee \psi$ iff $(p, i) \vDash \varphi$ or $(p, i) \vDash \psi$ and similarly $(p, i) \vDash \varphi \wedge \psi$ iff $(p, i) \vDash \varphi$ and $(p, i) \vDash \psi$

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 $(p, 1) \vDash \varphi$.

Definition 2.2. $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ consists of all positive existential FO sentences over the signature $\mathrm{Sch}_{\mathrm{Acc}}$.

Our main language of interest is $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$.

Example 2.3. [Benedikt et al. 2011; Calì and Martinenghi 2008] study query containment under (in our terminology, grounded) access patterns. Boolean query Q_1 is contained in Boolean query 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 [Calì and Martinenghi 2008] containment under access restrictions is shown to be decidable for conjunctive queries, while [Benedikt et al. 2011] 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 2.4. A boolean access AC_1 is said to be *long term relevant* [Benedikt et al. 2011] (LTR) for a boolean 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 [Benedikt et al. 2011] we say it is *LTR under grounded accesses* if there is a grounded access path satisfying the above.

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 to hold 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 access order restrictions and dataflow restrictions. For example, the following would restrict to paths in which names input to $\mathrm{Mobile}\#$ must have appeared previously in $\mathrm{Address}$:

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

Example 2.5. (Data integrity restrictions, continued) Let Sch be a schema that includes, in addition to the access methods, a set of functional dependencies (FDs) $d_i = R^i : pos_i \rightarrow a_i$, where pos_i are positions of R^i and a_i is a position of R^i. A database instance satisfies an FD above if any two tuples $\vec{s}, \vec{t}$ in R^i if $s_j = t_j$ for all $j \in pos_i$, then $s_{a_i} = t_{a_i}$. 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 2.4, but where each response returns only tuples in I.

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

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

where Q^{pre} and Q^{post} are defined as in the previous example. We will look at languages with inequalities in Section 6.

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. We emphasize again that we will be considering that satisfiability of formulas not arbitrary sequences of transitions, but only access paths. In such paths the instances change in a certain way: in a transition with an access on relation R, R_{pre} and R_{post} will change only on tuples consistent with the binding, while the other relations will not change at all. We will also consider satisfiability over grounded and exact paths, which impose restrictions which must be considered in any decision procedure for satisfaction or validity.

3. AN EXPRESSIVE LANGUAGE FOR ACCESS RESTRICTIONS

Since satisfiability for first-order logic is undecidable, it is clear that $\mathrm{AccLTL(FO)}$ 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. *Satisfiability of* $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ *is undecidable.*

This is surprising, as $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formulas are very restricted: they 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 only data that can be referenced across different configurations are the values in the binding, and the facts containing these values of these can only be related within consecutive configurations.

PROOF OF THEOREM 3.1. We first give an intuition over the proof of Theorem 3.1. The proof is a reduction from the problem of determining whether a collection Γ of functional dependencies and inclusion dependencies (IDs) implies another functional dependency σ. Functional dependencies are defined in Example 2.5 (see also [Abiteboul et al. 1995]). An inclusion dependency is given by a relation symbol R and sequence of positions $r_1 \dots r_k$ in R (that is, integers bounded by $arity(R)$), and a relation symbol S with positions $s_1 \dots s_k$. It holds in a database instance if for every tuple $t \in R$ there is a tuple $t' \in S$ such that $t_{r_i} = t'_{s_i}$ for all $i \leq k$. Since this problem is known to be undecidable [Chandra and Vardi 1985], it suffices to reduce it to unsatisfiability of an $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formula.

The difficulty here is that functional dependencies seem to require negation inside a universal quantification, while inclusion dependencies require quantifier alternation – in $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ we have only boolean combinations of positive sentences. We now explain the main idea involved in bridging this gap, which will also be used in later undecidability arguments (Theorem 6.2). The schema for our accesses includes a successor relation of a total order over the tuples of each relation mentioned 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 $\mathrm{Beg}(R)$ and $\mathrm{End}(R)$. Our formula will enforce 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. We now give the formal proof.

We assume that we are given a schema consisting of functional and inclusion dependencies Γ and a distinguished functional dependency σ. We can also assume all positions carry the same type. The undecidability of implication for IDs and FDs holds in the untyped setting [Abiteboul et al. 1995]. We will also verify the reduction over instances where all relations are of size at least two, allowing us to avoid certain corner cases.

We first give the schema produced by the reduction, which extends the relational schema Sch for the dependencies. For each relation R 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 is a relation $\mathrm{CheckIncDep}(R)$ 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.

Components of the sentence. We now build a sentence ψ as a conjunction of smaller sentences. For each relation R we have a subformula that describes a subpath that makes visible tuples in relations $\mathrm{Beg}(R), \mathrm{Succ}(R), \mathrm{End}(R)$ – one can also think of them as "building" these relations in the visible instance. The following sentence $\varphi_{\mathrm{next(R)}}$ describes a transition revealing a tuple $(\vec{t}_1, \vec{t}_2)$ in $\mathrm{Succ}(R)$ 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.

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

We will need to ensure the correctness of the reduction that for each LTrans path ρ satisfying the formula ψ the initial instance of ρ is empty. This can be ensured by the following sentence $\varphi_{\varnothing}$ =

$$
\bigwedge_{R \in \mathrm{Sch}} \neg(\exists \bar{x} \ R_{\mathrm{pre}}(\bar{x}))
$$

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

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

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

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

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

The sentence ψ we ultimately create will be the conjunction of $\varphi_\varnothing$ defined above with a set of sentences asserting that the path consists of subpaths satisfying

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

followed by a subpath satisfying $\varphi_{\mathrm{verify}}$, where $\varphi_{\mathrm{verify}}$, defined later, checks the constraints on each relation R.

$\varphi_{\mathrm{verify}}$ will not add tuples to the relations $\mathrm{Succ}(R)$ and $\mathrm{End}(R)$. Thus in any path that satisfies a sentence of the form above the visible instance associated to the position satisfying $\varphi_{\mathrm{verify}}$ must interpret $\mathrm{Succ}(R)$ 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 $\mathrm{End}(R)$. $\varphi_{\mathrm{verify}}$ will not reveal any more tuples in those relations, but instead will iterate over all pairs of tuples in the order given by $\mathrm{Succ}(R)$, performing additional accesses that will check that these tuples satisfy the dependencies in Γ and fail the distinguished dependency σ.

Invariant. Before giving additional detail on the conjunct $\varphi_{\mathrm{verify}}$, we now state the correctness property of the sentence ψ.

Given an access path p for this signature with final instance $J(p)$, we let $I(p)$ be the instance for the database schema containing only the relation R, with the interpretation of R having for each tuple $\vec{t}_1, \vec{t}_2$ in $J(p)(\mathrm{Succ}(R))$, the tuple $\vec{t}_1$ and also all tuples in $J(p)(\mathrm{End}(R))$.

Our first invariant will be: when this sentence is satisfiable by a path p, the structure $I(p)$ witnesses the failure of the implication of σ by Γ.

Conversely let I be a witness of the failure of the implication of σ by Γ. Let $<_I$ be a total order over the tuples of $I(R)$. We describe a path $p(I)$ of accesses and responses, from which we can infer a subinstance of the underlying instance. The path is built such that in the final instance of the path, the relation $J(p(I))(\mathrm{Succ}(R))$ represents the successor relation of $<_I$. To satisfy φ, p does an access first on the minimal tuple $\vec{t}$ of $<_I$ on the relation $\mathrm{Beg}(R)$, which will return true. Then starting from $\vec{t}$ and following the order $<_I$, the pairs $(\vec{t}_1, \vec{t}_2)$ such that $\vec{t}_2$ is the successor of $\vec{t}_1$ are exposed one by one by accesses to the relation $\mathrm{Succ}(R)$. The maximal tuple for $<_I$ is then exposed as being in $\mathrm{End}(R)$. Then the tuples of the relation $J(p(I))(\mathrm{CheckIncDep})$, which contains the same tuples as $I(R)$, are exposed one at a time following $<_I$. Finally the tuples of $J(p(I))(\mathrm{Chk}^{\mathrm{FD}})$, which are those in the cross product of $I(R)$ with itself, are exposed by accesses one by one.

Then our second invariant will be: for every counterexample I to the implication of σ by Γ, the path $p(I)$ satisfies the sentence.

Subformula performing the verification. $\varphi_{\mathrm{verify}}$ will be of the form

$$\varphi_{\mathrm{CheckFds}} \wedge \mathsf{X}(\varphi_{\mathrm{CheckIds}} \wedge \mathsf{X}\varphi_{\mathrm{CheckFdFailure}})$$

where $\varphi_{\mathrm{CheckFds}}$ checks that all functional dependencies in Γ hold, $\varphi_{\mathrm{CheckIds}}$ checks the inclusion dependencies, and $\varphi_{\mathrm{CheckFdFailure}}$ the failure of σ. We will focus on $\varphi_{\mathrm{CheckFds}}$ next, focusing on a single relation 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 $\mathrm{Succ}(R)$.

Let the functional dependencies on R in Γ be $fd_1 \ldots fd_j$. For each $fd_i \in \Gamma$ of the form $A \longrightarrow B$ we present a sentence $\varphi_{\mathrm{fd_i\text{-}tuple}}$ which checks that the current binding is a pair

$(\vec{t}_1, \vec{t}_2)$ which is not a counterexample to fd_i:

$$\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge$$
$$\neg(\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \Pi_A(\vec{t}_1) = \Pi_A(\vec{t}_2)) \vee$$
$$\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \Pi_B(\vec{t}_1) = \Pi_B(\vec{t}_2)$$

We now let $\varphi_{\text{allfd-tuple}}$ be

$$\varphi_{\text{fd}_1\text{-tuple}} \wedge \mathsf{X}(\varphi_{\text{fd}_2\text{-tuple}} \wedge \ldots \mathsf{X}\varphi_{\text{fd}_j\text{-tuple}})$$

We now consider an iterator $\varphi_{\text{checknext}}$ that checks all pairs. The sentence $\varphi_{\text{FD-init}}$ 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 sentence $\varphi_{FD-next}$ performs the iteration:

$$\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{post}}(\vec{t}_1, \vec{t}_2) \wedge$$
$$\neg(\exists \vec{t}_1 \vec{t}_2 \, \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{Chk}^{\mathrm{FD}}(R)_{\mathrm{pre}}(\vec{t}_1, \vec{t}_2)) \ \wedge (\varphi_1 \vee \varphi_2) \wedge \varphi_{\text{allfd-tuple}}$$

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 sentence states that the current position has an access to a pair $(\vec{t}_1, \vec{t}_2)$ that has not been checked before and such that: the access passes the test, and $Pred(\vec{t}_1, \vec{t}_2)$ 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 (as in φ_2).

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

$$\exists \vec{t}_1 \vec{t}_2 \, \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \mathrm{End}_{\mathrm{pre}}(\vec{t}_2) \wedge \varphi_{\text{allfd-tuple}}$$

We now construct the sentence $\varphi_{\text{CheckFds}}$ as $\varphi_{\text{FD-init}} \wedge \mathsf{X}\varphi_{\text{FD-next}} \mathsf{U} \varphi_{\text{FD-end}}$ 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 sentence $\varphi_{\text{CheckFdFailure}}$, checking that the distinguished functional dependency $\sigma = A' \longrightarrow B'$ fails, is simpler, since no iteration is needed:

$$\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \ \wedge \Pi_{A'}(\vec{t}_1) = \Pi_A(\vec{t}_2)) \wedge$$
$$\neg(\exists \vec{t}_1 \vec{t}_2 \ \mathrm{IsBind}_{\mathrm{Chk}^{\mathrm{FD}}(R)}(\vec{t}_1, \vec{t}_2) \wedge \Pi_B(\vec{t}_1) = \Pi_B(\vec{t}_2))$$

The formula $\varphi_{\text{CheckIds}}$ checking that an inclusion dependency, e.g. mapping positions $1 \ldots k$ of an R-tuple to positions $m_1 \ldots m_k$, is similar. The formula requires that the access path p make a single iteration over all individual tuples that have been revealed in the successor ordering, and for each such tuple u encountered, p must make some

access (e.g. to a new relation) with some tuple v, also appearing in $\mathrm{Succ}(R)$, such that $\bigwedge_i u_i = v_{m_i}$. Further details are spelled out in the proof of Theorem 6.2.

It is easy to check that the two invariants described earlier hold for this sentence.

□

4. VERIFIABLE SPECIFICATIONS: THE POSITIVE TRANSITION SUBLANGUAGE

The undecidability proof of $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ which adds an additional monotonicity condition. A $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$ formula φ is *binding-positive* if every atom of the form $\mathrm{IsBind}(\vec{w})$ 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$.

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:

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

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.*

4.1. Access Automata, their analysis, and connection with AccLTL^+

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 specifies 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 $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentence, while ψ^- is a positive boolean combination of negated $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences that cannot mention the predicate IsBind; all these formulas can use constants in the given set C.

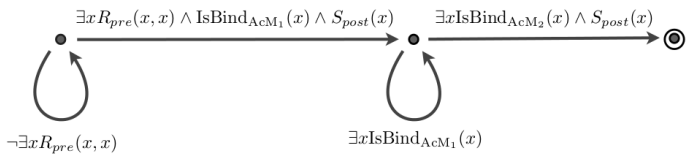

Fig. 2. An A-automaton.

Figure 2 shows an A-automaton. The automaton will stay in its initial state until an access is done using access method AcM_1 using a binding x such that $R(x,x)$ holds, at which point it will transition to a new state. After that the access path must consist of a sequence of accesses to AcM_1, terminating in an access on method AcM_2 with binding x, which must lead to a final configuration in which $S(x)$ holds.

Let $A = (S, s_0, F, \delta)$ be an A-automaton and let p be a path $t_1, \ldots, t_n$ 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 tuple 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 accepting. 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:

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.*

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 accepting state is s_1. For any disjointness constraint σ between R and S, let φ_σ be the following conjunctive query, which states that the disjointness constraint is violated.

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

Then the transitions are

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

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 $\neg Q'_{\mathrm{post}} \wedge Q_{\mathrm{post}}$.

□

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 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.

The proof relies on a conversion of propositional LTL formulas over finite words to finite state automata, plugging in the appropriate formulas for propositions.

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 (without embedded first order formulas). Let Φ be the set of maximal $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ subformulas of φ and let $\mathcal{P}$ be a set of propositions that contains a proposition P_χ for every $\chi \in \Phi$. We denote by $\tilde{\varphi}$ the LTL formula that is obtained from φ by replacing each subformula $\chi \in \Phi$ by P_χ. Note that $\tilde{\varphi}$ accepts words over the alphabet $2^{\mathcal{P}}$. 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 $2^{\mathcal{P}}$ in which S_i is the set of all predicates P_χ such that χ holds on $(\mathrm{I}_i, \mathrm{AC}_i, \mathrm{I}'_i)$, $i \leq n$. With these definitions, the following proposition is obvious:

PROPOSITION 4.6. *For every* AccLTL^+ *formula* φ *and every access path* p, φ *holds on* p *iff* $\tilde{\varphi}$ *holds on* $\tilde{p}$.

We will call a propositional symbol P_χ *even* in $\tilde{\varphi}$ if each occurrence of P_χ in $\tilde{\varphi}$ is under an even number of negations. We will later exploit 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 $\tilde{\varphi}$. We next show that $\tilde{\varphi}$ has a certain monotonicity property with respect to even symbols.

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

PROPOSITION 4.7. *Let* φ *be an LTL formula and let* P *be an even propositional symbol in* φ. *Then for all words* w *over* $2^{\mathcal{P}}$ *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 $w, j \vDash \varphi$. 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. □

Combining the previous proposition with a standard construction taking propositional temporal logic (see the online appendix), we get that $\tilde{\varphi}$ can be converted to a finite automaton:

PROPOSITION 4.8. $\tilde{\varphi}$ *is equivalent (or all words over* $2^{\mathcal{P}}$*) to a non-deterministic automaton* M *of size at most exponential in* $\tilde{\varphi}$. *In addition,* M *can be taken to have the property: if* (p, S, q) *is a transition in* M *and if* P *is even in* $\tilde{\varphi}$, *then* $(q, S \cup \{P\}, q)$ *is a transition in* M.

In the next step we modify M by simply replacing the propositional formulas by the first order formulas they represented, reversing the abstraction process used to go from φ to $\tilde{\varphi}$. This process will produce a mild variation of our prior automata model for access paths that we define next, which we refer to these as FO-automata; these

are simply a generalization of A-automata that do not have restrictions concerning formulas containing the predicate $\mathrm{IsBind}_{\mathrm{AcM}}$. Formally, an ***FO-automaton*** consists of a finite set of states Q, an initial state $q_0 \in Q$, a set $F \subseteq Q$ of accepting states, and a set δ of transitions of the form (p, φ, q) where $p, q \in Q$ and φ is a boolean combination of $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ sentences. The notions of (accepting) runs and accepted language are exactly as those for A-automata.

We create an FO automaton F from M as follows: F has the same states as M, and the initial and accepting states are also the same. The transitions of F are obtained from the transitions of M by replacing each set S on a transition by the formula $\psi^- \wedge \psi^+$ where

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

From the definition of $\tilde{\varphi}$, on can se that for every access path p, φ holds on p iff M accepts $\tilde{p}$.

In general F will contain transitions that are labelled by subformulas $\psi \wedge \neg\psi'$ where $\psi' \in \Phi$ is an $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula that contains the predicate $\mathrm{IsBind}_{\mathrm{AcM}}$. Recall that by definition of an A-automaton the predicate $\mathrm{IsBind}_{\mathrm{AcM}}$ can only occur under an even number of negations. Hence, F is not an A-automaton in general. However, we now show that F is equivalent to an A-automaton A. Let $(p, \psi \wedge \neg\psi', q)$ be a transition of F for which $\psi' \in \Phi$ is an $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula that contains the predicate $\mathrm{IsBind}_{\mathrm{AcM}}$. Recall that we argued previously that in this case $P_{\psi'}$ must be even. Hence, it follows from Proposition 4.8 and from the way that F is obtained from M that F also contains the transition $(p, \psi \wedge \psi', q)$. Thus if we replace the transition $(p, \psi \wedge \neg\psi', q)$ by (p, ψ, q) we do not change the accepted language. Repeating this transformation, we obtain an A-automaton A equivalent to F.

We still need to verify that A has size at most exponential in φ. This is the case because the only blow-up in the above construction occurs in Proposition 4.8.

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.

THEOREM 4.9. *Emptiness of A-automata is decidable in 2EXPTIME. The same holds if accesses are restricted to be exact or idempotent.*

Notice that from Theorem 4.9 and Proposition 4.4 we get a 2EXPTIME upper bound for containment and long-term relevance. This improves on the prior known bounds [Benedikt et al. 2011; Calì and Martinenghi 2008].

In the body of the paper, we give the proof only for general paths. The discussion of how to extend to exact and idempotent accesses is left to the online appendix. 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 4.10. *Emptiness of A-automata and satisfiability of* AccLTL^+ *are both* 2*EXPTIME-hard.*

These results are the most involved in the paper, and will be developed over the next few subsections.

We reduce the emptiness problem for A-automata to the problem of whether a Datalog program is contained within a positive first-order query. The reduction to this problem in turn involves several stages, and the first step goes through a syntactic subclass of A-automata, called “linear 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 linear A-automata. We will prove that the emptiness of a linear A-automaton can be reduced to the containment of a datalog program into a positive sentence.

4.2. First step: Reduction to A-Automata of a special form

We now present the reduction of non-emptiness test for A-automata to a simpler automaton model called *linear A-automaton*. The binding-positive restriction plays a role in the reduction of the emptiness of an A-automaton to the emptiness of linear A-Automata. This restriction also helps in the reduction of the nonemptiness problem for linear A-automaton to the containment problem for a Datalog program in a positive sentence.

Before giving the definitions, we need to present some notation. Let φ be a formula over a schema Sch. Then by $\mathrm{post}(\varphi)$ we denote the formula obtained from φ by subscripting the relations of φ by post.

Let ψ be a sentence over a schema Sch such that ψ is a conjunction of a positive sentence and a positive Boolean combination of negated positive sentences. We denote by (i) ψ^+ the positive sentence in the conjunction ψ, (ii) ψ^- the positive Boolean combination of negated positive sentences in the conjunction ψ. The same notation is used for sentences in $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$.

Let θ be the conjunction of a positive Boolean combination of negated positive sentences in $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ which have no predicates $\mathrm{IsBind}_{\mathrm{AcM}}$ and of a positive sentence in $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$. We denote by $\mathrm{Sch}(\theta)$ the formula over Sch obtained by replacing any relation name R_{pre} or R_{post} by the relation name R and by replacing the atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{x})$ by True. We can observe that $\mathrm{Sch}(\theta)$ is a conjunction of a positive sentence and a positive Boolean combination of negated positive sentences; furthermore, $\mathrm{Sch}(\theta^+) = \mathrm{Sch}(\theta)^+$ and $\mathrm{Sch}(\theta^-) = \mathrm{Sch}(\theta)^-$.

Let LTrans be the LTS associated with Sch. Let ψ and φ be two sentences in $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$. *$\psi$ implies φ over* LTrans iff for each LTrans transition t, if t satisfies ψ then t satisfies φ. The main difference from ordinary implication is that we consider only LTrans transitions $(I, (\mathrm{AcM}, \bar{b}), I')$. For valid transitions we always have I is included in I'. For example the formula $\psi = \exists x R_{\mathrm{pre}}(x) \wedge S_{\mathrm{post}}(x)$ does imply the formula $\varphi = \exists x R_{\mathrm{post}}(x) \wedge S_{\mathrm{post}}(x)$ over LTrans; however, ψ does not imply ψ over general instances over $\mathrm{Sch}_{\mathrm{Acc}}$.

Definition 4.11 (*Linear A-automaton*). An A-automaton A is *linear* iff the following conditions hold:

(Singleton). Every strongly connected component (SCC) of A consists of only one state.
(Deterministic-Access-Method). For any transition $(s, \varphi, s') \in \delta$, φ^+ is a formula of the form $\exists \bar{x}\ \mathrm{IsBind}_{\mathrm{AcM}}(\bar{x}) \wedge \psi(\bar{x})$ where $\psi(\bar{x})$ is a formula of $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ without any atom of the form $\mathrm{IsBind}_{\mathrm{AcM}'}(\bar{y})$.
(Deterministic-Transition).

- The states of A are linearly ordered by the reachability relation. That is, they can be ordered as $s_1, \ldots, s_h$, such that for each $i < h$, there is exactly one transition from the state s_i to s_{i+1}.
- Note that from requirement (Deterministic Access Method), the formula φ^+ is of the form $\exists \bar{x}\, \mathrm{IsBind}_{\mathrm{AcM}}(\bar{x}) \wedge \psi(\bar{x})$ such that ψ is a positive formula of $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ which does not have any atom of the form $\mathrm{IsBind}_{\mathrm{AcM}'}(\bar{y})$. We further require that for each transition (s_i, ψ, s_{i+1}) of A, there exists a vector of constants $\bar{c}$ and a sentence φ such that $\psi(\bar{x}) = (\bar{x} = \bar{c}) \wedge \varphi$.

Therefore, φ^+ is equivalent to the formula $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{c}) \wedge \varphi$. We recall that φ is a $\mathrm{FO}^{\exists+}_{\mathrm{Acc}}$ formula having no predicates $\mathrm{IsBind}_{\mathrm{AcM}'}$.

(Negative-Invariant). For each state s, there exists a positive Boolean combination of negated positive sentences over Sch, denoted by $\Omega(s)$, such that for each transition (s', ψ, s) of A, ψ^- implies $\mathrm{post}(\Omega(s))$ over LTrans. If $s' = s$, then we also require that $\mathrm{post}(\Omega(s))$ implies ψ^- over LTrans.

(Initial-Final-State). The initial state is the state s_1 and the accepting state is s_h.

The number h is called the *height* of A and is denoted by $\mathrm{height}(A)$.

The first few requirements state that the automaton consists of a line of states with self-loops, with the transitions that change states using constants symbols, and each transition fixing an access method. Condition (Negative Invariant) imposes that the negative properties in transition formulas, which a priori mention the pre- and post-version of the transition, can be reduced to enforcing that some negative properties hold invariantly on all instances visible while in the state. Restating it slightly, it states that each state s of a linear A-automaton A is associated with a positive Boolean combination of negated positive sentences, $\Omega(s)$, such that:

(NegativeInv +) If (s, ψ, s) is a transition of A and instance I satisfies $\Omega(s)$, then for any LTrans transition $t = (I', (\mathrm{AcM}, \bar{b}), I'')$, if $I' \subseteq I'' \subseteq I$ then t satisfies ψ^-.

(NegativeInv †) If (s, ψ, s') is a transition of A and an LTrans transition $(I, (\mathrm{AcM}, \bar{b}), I')$ satisfies ψ then I' satisfies $\Omega(s)$.

The following lemma shows that an A-automaton corresponds, up to emptiness, to unions of linear A-automata.

LEMMA 4.12. *Let A be an A-automaton. Then, there exists a set $\{A_i\}_{1 \le i \le n}$ of linear A-automata such that for each $1 \le i \le n$, A_i has size polynomial in the size of A, n is exponential in A and $L(A)$ is empty iff $L(A_1) \cup \ldots \cup L(A_n)$ is empty.*

The statement hold both for the general path semantics and for restricted (exact/idempotent) paths. The proof is presented in the online appendix. Note that Condition (Deterministic-Access-Method) can be assured by simply breaking up a single transition into multiple transitions, each of which commits to a single access.

4.3. From Emptiness of linear A-automata to Containment of Datalog in Positive Queries: statement of main results

We now proceed to show that emptiness of a linear A-automata is decidable. Together with Lemma 4.12, this implies the decidability of emptiness for (general) A-automata. This will involve reducing the emptiness of a linear 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*, denoted by $\mathrm{Sch}_{\mathrm{ext}}$ and the *intentional schema*, denoted by $\mathrm{Sch}_{\mathrm{int}}$. A *Datalog program* $\mathcal{P}$ is a finite set of rules of the form "$\mathrm{head} :- \mathrm{body}$" where head is an atomic formula $R(\bar{x})$ with a relation symbol R in the intentional schema, and where body is a conjunctive query that can use relation symbols from the intentional and the extensional schema. Each Datalog program $\mathcal{P}$ contains a distinguished *goal predicate*. We use the standard notions of the least fixed point of a Datalog program $\mathcal{P}$ on a database D (see [Abiteboul et al. 1995]), and we denote this fixed point by $\mathcal{P}^{full}(D)$. We say that a Datalog program $\mathcal{P}$ *accepts* a database D if the goal predicate of $\mathcal{P}$ is not empty in $\mathcal{P}^{full}(D)$.

LEMMA 4.13. *Let A be a linear 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 complex, and will take up the bulk of the remainder of the section.

Let $A = (S, s_1, \{s_h\}, \delta)$ be a linear A-automaton, with the set of states $S = \{s_1, \ldots, s_h\}$, where the indexing representing a linear-order of the single-state components.

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 linear automaton, a run always proceeds through the same sequence of states. Let ρ be an LTrans path and r be an accepting run on ρ. The extensional database D will have predicates $\mathrm{BackgroundR}_i$ representing the part of relation R that becomes visible when doing a transition from the i^{th} state to itself and predicates $\mathrm{IntBackgroundR}_i$ representing the data that becomes visible when crossing from the i^{th} state to the $(i+1)^{st}$ state. The important intentional predicates ViewR_i will represent intermediate stages of the predicates $\mathrm{BackgroundR}_i$ within the evolution of each state. The Datalog program $\mathcal{P}_A$ will have rules corresponding to the evolution of ViewR_i by adding tuples from $\mathrm{BackgroundR}_i$ and $\mathrm{IntBackgroundR}_{i-1}$. To ensure that the tuples in ViewR_i 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 to 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. We give the formal definition of $\mathcal{P}_A$ and $\mathcal{P}'_A$ in Subsection 4.4 below.

In the proof that our construction is correct, we show that the Datalog program $\mathcal{P}_A$ can be decomposed into subprograms $\mathcal{P}_1, \ldots, \mathcal{P}_h$ that correspond to the states $s_1, \ldots, s_h$ of A in the following sense: Whenever an A-automaton has a run that ends in state s_i, $i \leq h$ then the subprogram $\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_i$ of $\mathcal{P}$ adds tuples to the intentional database that correspond in a certain way to the tuples that A has obtained using accesses while arriving at s_i. The formal proof is in Subsection 4.5 below.

4.4. The reduction from A-automata non-emptiness to query containment: Construction of the Datalog Program and the Positive Query

We give the construction of the Datalog program $\mathcal{P}_A$ and the positive query $\mathcal{P}'_A$ required by Lemma 4.13.

4.4.1. The Program $\mathcal{P}_A$. We 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 arity of R. Finally, $\mathrm{Sch}_{\mathrm{ext}}$ contains for each access method AcM, a relation $\mathrm{Bind}_{\mathrm{AcM}}$ of arity equal to the size of $\mathrm{Inp}(\mathrm{AcM})$.

The intentional 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 predicate ViewR_i whose arity is the arity of R;
— for every $1 \leq i \leq \mathrm{height}(A)$, a predicate ReachC_i of arity of 0;
— for each $R \in \mathrm{Sch}$, $1 \leq i \leq \mathrm{height}(A)$, for each access method AcM such that $\mathrm{Rel}(\mathrm{AcM}) = R$, a predicate $\mathrm{Answer}(\mathrm{AcM})_i$ with arity equal to the sum of the arity of R and the size of $\mathrm{Inp}(\mathrm{AcM})$; and also a predicate $\mathrm{IntAnswer}(\mathrm{AcM})_{i-1}$ with arity equal to the sum of the arity of R and the size of $\mathrm{Inp}(\mathrm{AcM})$.
— for each access method $\mathrm{AcM} \in \mathrm{Sch}$ and $1 \leq i\,\mathrm{height}(A)$, a predicate $\mathrm{Done\text{-}Access}^i_{\mathrm{AcM}}$ and a predicate $\mathrm{IDone\text{-}Access}^i_{\mathrm{AcM}}$ both of arity equal to the size of $\mathrm{Inp}(\mathrm{AcM})$

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

Recall that an access path of a linear Automaton of height h will break up into $h-1$ transitions which correspond to changing components, plus h subpaths representing looping within a component. The Datalog program conjoined with negation of the UCQ will accept databases that code such paths.

— Any relation R in the background database that the A-automaton A accesses will be equal to the union of the relations $\mathrm{BackgroundR}_1, \ldots, \mathrm{BackgroundR}_h$, $h = \mathrm{height}(A)$, along with the relations $\mathrm{IntBackgroundR}_0, \ldots, \mathrm{IntBackgroundR}_{h-1}$. The tuples in $\mathrm{BackgroundR}_i$, $i \leq h$, are those that the background database might return when A accesses relation R whilst remaining in the same state s_i. The tuples in $\mathrm{IntBackgroundR}_0$ are the tuples of the initial instance. The tuples in $\mathrm{IntBackgroundR}_i$, $0 < i < h$, are those that the background database returns when A accesses relation R whilst crossing from s_i into state s_{i+1}.
— The relation $\mathrm{Bind}_{\mathrm{AcM}}$ contains the tuples $\bar{b}$ such that $(\mathrm{AcM}, \bar{b})$ is an access done in the LTrans path accepted by A.
— The union of the relations $\mathrm{ViewR}_1, \ldots, \mathrm{ViewR}_i$ corresponds to the relation R in the instance that A stores internally when it is in s_i. Thus each ViewR_i will be derived as a union of selections from suitable $\mathrm{BackgroundR}_j$ and $\mathrm{IntBackgroundR}_j$.
— The relation $\text{Done-Access}_{\mathrm{AcM}}$ contains the tuples $\bar{b}$ such that $(\mathrm{AcM}, \bar{b})$ is an access leading the path from a state s_i to the state s_i.
— The relation $\text{IDone-Access}_{\mathrm{AcM}}$ contains the binding leading the path from a state s_{i-1} to s_i.
— The predicate ReachC_i indicates that there is a path ρ of LTrans and a run r of A that starts in the initial state and ends in the state s_i such that r is a run of A on ρ.
— A tuple $\bar{b}\cdot\bar{t}$ is in $\mathrm{Answer}(\mathrm{AcM})_i$ iff $\bar{t}$ is in $\mathrm{BackgroundR}_i$ and $\{\bar{t}\}$ is a well-formed output with the access $(\mathrm{AcM}, \bar{b})$. A tuple $\bar{b}\cdot\bar{t}$ is in $\mathrm{IntAnswer}(\mathrm{AcM})_i$ iff $\bar{t}$ is in $\mathrm{IntBackgroundR}_i$ and $\{\bar{t}\}$ is a well-formed output with the access $(\mathrm{AcM}, \bar{b})$.

In this Datalog program, we allow the use of "extended rules" where the body $\varphi(\vec{x})$ is a positive query. We require that the extended rules use in their body a *guarded safe positive formula*; a formula $\varphi(\bar{x})$ that is the conjunction of a positive formula $\varphi_2(\bar{x})$ with a conjunctive query $\varphi_1(\bar{x})$ such that every free variable of $\varphi_2(\bar{x})$ is included in some relational atom of $\varphi_1(\bar{x})$. We allow a vacuous case consisting of a positive formula with no free variables. The extension of the semantics of Datalog to this case is obvious. One can also see that this class of formulas is closed under conjunction.

Rules for predicates ViewR_i. We now define the rule for the predicates ViewR_i. For $i > 0$, for each relation $R \in \mathrm{Sch}$, for each access method AcM such that $\mathrm{Rel}(\mathrm{AcM}) = R$,

$$\mathrm{ViewR}_i(\bar{z}) \ :-\ \text{Done-Access}^i_{\mathrm{AcM}}(\bar{x}), \mathrm{Answer}(\mathrm{AcM})_i(\bar{x}, \bar{z})$$

$$\mathrm{ViewR}_i(\bar{z}) \ :-\ \text{IDone-Access}^i_{\mathrm{AcM}}(\bar{x}), \mathrm{IntAnswer}(\mathrm{AcM})_i(\bar{x}, \bar{z})$$

For each relation $R \in \mathrm{Sch}$,

$$\mathrm{ViewR}_0(\bar{x}) \ :-\ \mathrm{IntBackgroundR}_0(\bar{x})$$

Rules for predicates $\text{Done-Access}^i_{\mathrm{AcM}}$. We consider two kinds of transitions: the loop transitions going from one state to itself and the crossing transitions going from one state to another state.

Rule for loop transitions. Let $d = (s_i, \psi^d, s_i)$ be a loop transition. Due to Condition (Deterministic-Access-Method), there exists an access method AcM such that ψ^{d^+} is

equal to $\exists \bar{x}\, \mathrm{IsBind}_{\mathrm{AcM}}(\bar{x}) \wedge \rho^d(\bar{x})$, where ρ^d does not contain any atom with relation name $\mathrm{IsBind}_{\mathrm{AcM}'}$.

The formula $\mathrm{ValidBind}_d(\bar{x})$, where $\bar{x}$ are the free variables, is obtained from ψ^d as follows:

— The atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{x})$ is replaced by $\mathrm{Bind}_{\mathrm{AcM}}(\bar{x})$.
— Each atom $T_{\mathrm{pre}}(\bar{y})$ is replaced by $\bigvee_{j \leq i} \mathrm{ViewT}_j(\bar{y})$, where s_i is the state referenced in d
— Each atom $T_{\mathrm{post}}(\bar{y})$ such that T is different from R is replaced by $\bigvee_{j \leq i} \mathrm{ViewT}_j(\bar{y})$, where s_i is the state referenced in d.
— Each atom $R_{\mathrm{post}}(\bar{y})$ is replaced by $\mathrm{Answer}(\mathrm{AcM})_i(\bar{x}, \bar{y}) \vee \bigvee_{j \leq i} \mathrm{ViewT}_j(\bar{y})$, where s_i is the state referenced in d.

Notice that $\mathrm{ValidBind}_d$ is of the form $\exists\, \mathrm{Bind}_{\mathrm{AcM}}(\bar{x}) \wedge \gamma(\bar{x})$, and is thus a guarded safe positive formula.

Claim 1 presented in Subsubsection 4.5.1 explains that a tuple $\bar{b}$ appears in $\mathrm{ValidBind}_d$ iff there exists a LTrans transition $t = (I, (\mathrm{AcM}, \bar{b}), I')$ satisfying ψ^+ such that for each relation name T in Sch, $I(T)$ contains all the tuples in $\bigcup_{k \leq i} \mathrm{ViewT}_k$ and for each relation name T different from R, $I'(T)$ contains the tuples in $I(T)$ and $I'(R)$ contains the tuples in $I(R)$ along with any tuples in the maximal well-formed output for $(\mathrm{AcM}, \bar{b})$ on the relation R accessed by AcM, interpreted as in $D(\mathrm{BackgroundR}_i)$.

We are now ready to give the rule associated with this transition from s_i to itself:

$$\mathrm{Done\text{-}Access}^i_{\mathrm{AcM}}(\bar{x}) \text{ :- } \mathrm{ReachC}_i, \mathrm{ValidBind}_d(\bar{x})$$

Since $\mathrm{ValidBind}_d$ is a guarded safe positive formula, the rule uses a guarded safe positive formula.

Rule for crossing transitions. Let $i > 2$. Let $d = (s_{i-1}, \psi, s_i)$ be a transition. Due to Condition (Deterministic-Transition), there exists one access denoted by $(\mathrm{AcM}, \bar{b})$ such that ψ^+ is equal to $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{b}) \wedge \psi_1$. The rule associated to d is

$$\mathrm{IDone\text{-}Access}^i_{\mathrm{AcM}}(\bar{b}) \text{ :- } \mathrm{ReachC}_i$$

The body formula of this rule is a conjunctive query, and thus clearly a guarded safe positive formula.

Rule for ReachC_i. The predicate ReachC_1 is always true.

Let i be an integer strictly greater than 1. Let $d = (s_{i-1}, \psi^d, s_i)$ be a transition. Due to Condition (Deterministic-Transition), there exists one access, denoted by $(\mathrm{AcM}, \bar{b})$, such that the positive part $(\psi^d)^+$ is equal to $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{b}) \wedge \rho^d$ for some ρ^d. We define the predicate ReachC_i to be true iff the sentence $\mathrm{IsValidBinding}_d$ is true and the predicate ReachC_{i-1} is true, where the sentence $\mathrm{IsValidBinding}_d$ is obtained from $(\psi^d)^+$ by performing the replacements below:

— the atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{x})$ is replaced by $\mathrm{Bind}_{\mathrm{AcM}}(\bar{b})$,
— each atom $T_{\mathrm{pre}}(\bar{y})$ is replaced by $\bigvee_{j<i} \mathrm{ViewT}_j(\bar{y})$,
— each atom $T_{\mathrm{post}}(\bar{y})$ where T is different from R is replaced by $\bigvee_{j<i} \mathrm{ViewT}_j(\bar{y})$,
— each atom $R_{\mathrm{post}}(\bar{y})$ is replaced by $\mathrm{IntAnswer}(\mathrm{AcM})_{i-1}(\bar{b}, \bar{y}) \vee \bigvee_{j<i} \mathrm{ViewR}_j(\bar{y})$, where R is equal to $\mathrm{Rel}(\mathrm{AcM})$.

Due to the fact that ψ^{d^+} is a sentence, $\mathrm{IsValidBinding}_d$ is a guarded safe positive formula.

It will turn out (see Claim 2) that sentence $\mathrm{IsValidBinding}_d$ is true on an input instance D for the Datalog program iff the access transition $(I, (\mathrm{AcM}, \bar{b}), I')$ satisfies ψ^{d^+}, where for each $R \in \mathrm{Sch}$, the tuples in $I(R)$ are those in any of the sets of tuples

$\mathcal{P}_A^{full}(D)(\mathrm{ViewR}_k)$, for k less or equal to i and $I'(R)$ is the union of $I(R)$ and the well-formed output of $(\mathrm{AcM}, \bar{b})$ on $D(\mathrm{IntBackgroundR}_{i-1})$.

Rules for $\mathrm{IntAnswer(AcM)}_i$ ***and*** $\mathrm{Answer(AcM)}_i$. Let i be an integer greater than 0, R be a relation of Sch and AcM an access method of Sch such that $\mathrm{Rel(AcM)} = R$. Let $\bar{y}$ and $\bar{x}$ be two vectors of variables such the length of $\bar{y}$ is equal to the arity of R and the length of $\bar{x}$ is equal to the size of $\mathrm{Inp(AcM)}$. We say $\bar{y} = y_1 \dots y_e$ is *compatible* with AcM and $\bar{x}$ if $\bar{x}$ consists of the y_i corresponding to input positions of AcM in increasing order.

We have the rules:

$$\mathrm{IntAnswer(AcM)}_i(\bar{x}, \bar{y}) \ :- \ \mathrm{IntBackgroundR}_i(\bar{y})$$

where $\bar{y}$ is compatible with AcM and $\bar{x}$.

$$\mathrm{Answer(AcM)}_i(\bar{x}, \bar{y}) \ :- \ \mathrm{BackgroundR}_i(\bar{y})$$

where $\bar{y}$ is compatible with AcM and $\bar{x}$.

4.4.2. The positive query $\mathcal{P}'_A$. We now give the construction of the positive query $\mathcal{P}'_A$. Let i be an integer less than h. Let Φ_i^- be the smallest set containing the following formulas:

— Φ_i^- contains the formula $\Omega_{Datalog}(\Omega(s_i))$ that is obtained from $\Omega(s_i)$ by replacing each atom $R(\bar{x})$ by

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

— for the transition (s_{i-1}, φ, s_i) of A, Φ_i^- contains the formula $\Omega_{Datalog}(\varphi^-)$ 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 $\mathrm{dual}(\bigwedge_{i \leq h} \Phi_i^-)$ where $h = \mathrm{height}(A)$. 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. Thus a structure that satisfies the negation of the positive query will need to satisfy the translation of the negative invariants to the Datalog schema.

4.4.3. From "extended Datalog program" to Datalog program. We finish this subsection with a lemma explaining that allowing positive formulas in the body of the rules of a Datalog program was only syntactic sugar.

LEMMA 4.14. *Let $\mathcal{P}$ be an "extended Datalog program", allowing guarded safe positive formulas, even allowing equality predicates in the body of the rules of $\mathcal{P}$. Then there exists a classical Datalog program $\mathcal{P}'$ of size polynomial in the size of $\mathcal{P}$ such that for any instance I, I satisfies $\mathcal{P}$ iff I satisfies $\mathcal{P}'$.*

PROOF SKETCH. First, we remove the equality predicate from the extensional schema and we add a corresponding predicate Equal to the intentional schema. The only rule associated with the intentional relation Equal is $\mathrm{Equal}(x, x) \ :- \ \mathrm{Adom}(x)$. This rule uses a new intentional predicate Adom, and we thus add rules to define

this. Intuitively, all the values belonging to an instance appear in the relation Adom. The rules for the relation Adom are of the form $\mathrm{Adom}(x_j)$:– $\mathrm{BackgroundR}_i(\bar{x})$ or $\mathrm{Adom}(x_j)$:– $\mathrm{IntBackgroundR}_i(\bar{x})$ or $\mathrm{Adom}(x_j)$:– $\mathrm{Bind}_{\mathrm{AcM}}(\bar{x})$ where i is less than h and j is an integer. For each constant c in $\mathcal{C}$, the rule $\mathrm{Adom}(c)$:– is added. The remainder of the lemma follows from the folklore theorem that any guarded safe positive query can be rewritten into an non-recursive equivalent Datalog program of size polynomial in the size of the positive query. □

4.5. From the containment of the Datalog program in the positive query to emptiness of the linear progressive automaton

We consider the Datalog program $\mathcal{P}_A$ constructed from the linear A-automaton A in Lemma 4.13. We will show that if there is a database D that is a model of $\mathcal{P}_A$ but not of $\mathcal{P}'_A$, then A has an accepting run over the database instance $\mathrm{Sch}(D)$ for the original schema Sch, where $\mathrm{Sch}(D)$ is such that for each relation $R \in \mathrm{Sch}$, the tuples in $\mathrm{Sch}(D)(R)$ are those in $\bigcup_{k \leq height(A)} D(\mathrm{ViewR}_k)$.

This subsection is split in four parts. The first part makes the link between the satisfaction of the formula appearing in the transitions of A and the formula appearing in the rule of $\mathcal{P}_A$ and $\mathcal{P}'_A$. The second part explains that the $\mathcal{P}_A$ can be decomposed into an equivalent sequence of datalog programs. The third part is concerned with the construction of partial LTrans paths associated with each of the subprograms built in the previous part. The last part explains how to use the partial LTrans paths to obtain an accepting LTrans path.

4.5.1. Correctness of translations between formula from A-automaton and Datalog Program. The first claim is concerned with the relationship between the formula ψ^+ in a self-loop transition $d = (s_i, \psi, s_i)$ and the formula $\mathrm{ValidBind}_d$.

We first need some notation. For a database I and relation R we let $R_{\mathrm{AcM}}(\bar{b})(I)$ denote the maximal well-formed answer for the access $(\mathrm{AcM}, \bar{b})$ on I where $\mathrm{Rel}(\mathrm{AcM}) = R$. The set of facts in R composed of the tuples in $R_{\mathrm{AcM}}(\bar{b})(I)$ is denoted by $(\mathrm{AcM}, \bar{b}, I)$.

Given database K over schema Sch, an integer i and a set of facts M of the relation R, we define

— by $K + M$ the instance over Sch such that:
  — for each T different from R, the tuples in $K + M(T)$ are those in $K(T)$,
  — the tuples in $K + M(R)$ are those in $K(R)$ along with those associated with the facts in M.

CLAIM 1. *Let:*

— Sch *be a schema,*
— A *be a linear A-automaton,*
— $d = (s_i, \psi, s_i)$ *be a transition of A,*
— AcM *be the access method associated with* ψ *following Condition (Deterministic-Access-Method),*
— $\bar{b}$ *be a binding of* AcM*,*
— $\mathrm{ValidBind}_d$ *be the formula obtained from* d *in the construction of* $\mathcal{P}_A$*,*
— I *be an instance of* Sch *and* $K_1, \cdots, K_i$ *be a set of instances of* Sch *such that* $K_i \subseteq I$*,*
— K *be the union of* K_j *(that is, each relation symbol in* K *is interpreted by the union of the interpretations in* K_j*),*
— D *be the database instance for the Datalog schema in which (i) each relation* ViewR_j*,* $j \leq i$*, of* D *is interpreted as in the relation* R *of* K_j*; (ii) each relation* $\mathrm{Answer}(\mathrm{AcM}')_i$ *is interpreted by the set of tuples* $\bar{b}' \cdot \bar{t}$ *where* $\bar{t}$ *ranges over tuples in* $R_{\mathrm{AcM}'}(\bar{b}')(I)$ *for each access* $(\mathrm{AcM}', \bar{b}')$ *where* $R = \mathrm{Rel}(\mathrm{AcM}')$*; (iii) the relation* $\mathrm{Bind}_{\mathrm{AcM}}$ *is interpreted by the singleton tuple* $\bar{b}$

Then ψ^+ *is satisfied by* $(K, (\mathrm{AcM}, \bar{b}), K + (\mathrm{AcM}, \bar{b}, I))$ *iff* $\mathrm{ValidBind}_d(\bar{b})$ *is satisfied by* D.

The proof of this claim is from unwinding the definition of $\mathrm{ValidBind}_d$. For completeness, it is proven in detail in the online appendix.

The second claim analogously gives the relationship between the formula ψ^+ of a transition d of the form (s_{i-1}, ψ, s_i) (that is, between components) and the formula $\mathrm{IsValidBinding}_d$.

CLAIM 2. *Let:*
— Sch *be a schema,*
— A *be a linear A-automaton,*
— $d = (s_{i-1}, \psi, s_i)$ *be a transition in* A *from* s_{i-1} *to* s_i*,*
— AcM *and* $\bar{b}$ *be the unique access method and the binding associated with* ψ^+ *following Condition (Deterministic-Transition),*
— $\mathrm{IsValidBinding}_d$ *be the formula built from* d *in the construction of* $\mathcal{P}_A$*,*
— I *be an instance of* Sch *and* $K_1, \cdots, K_{i-1}$ *be a set of instances of* Sch*,*
— K *be the union of the* K_k*,*
— D *be a database instance for the Datalog Schema in which (i) each relation* ViewR_j*,* $j \leq i-1$*, of* D *is interpreted as in the relation* R *of* K_j*; (ii) each relation* $\mathrm{IntAnswer(AcM)}_{i-1}$ *is interpreted by the set of the tuples* $\bar{b} \cdot \bar{t}$ *where* $\bar{t}$ *is a tuple in* $R_{\mathrm{AcM}}(\bar{b})(I)$*; (iii) the relation* $\mathrm{Bind}_{\mathrm{AcM}}$ *contains only the tuple* $\bar{b}$

Then ψ^+ *is satisfied by* $(K, (\mathrm{AcM}, \bar{b}), K + (\mathrm{AcM}, \bar{b}, I))$ *iff* $\mathrm{IsValidBinding}_d$ *is satisfied by* D.

The proof of this claim also comes from simply unwinding the definition of $\mathrm{ValidBind}_d$, and is analogous to the proof of the previous claim.

For the next two claims, we need the following definition. Let Sch be a schema, D be an instance over $\mathrm{Sch}_{\mathrm{ext}}$ and i and j be integers. Let $I_{D,i,j}$ be the instance over Sch such that for each relation name R, the tuples in $I_{D,i,j}(R)$ are the tuples in $\bigcup_{m \leq i} D(\mathrm{BackgroundR}_m) \cup \bigcup_{k \leq j} D(\mathrm{IntBackgroundR}_k)$.

The third claim is concerned with the relationship between the satisfaction of $\Omega(s_i)$ and the satisfaction of $\Omega_{Datalog}(\Omega(s_i))$. We recall that $\Omega(s_i)$ is a Boolean combination of negated positive sentences over Sch, where s_i is the i^{th} state of A. The sentence $\Omega_{Datalog}(\Omega(s_i))$ is obtained from $\Omega(s_i)$ by replacing each atom $R(\bar{x})$ by

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

CLAIM 3. *Let:*
— Sch *be a schema,*
— A *be a linear A-automaton,*
— s_i *be the* i^{th} *state of* A*,*
— D *be an instance over* $\mathrm{Sch}_{\mathrm{ext}}$

Then $\Omega(s_i)$ *is satisfied by* $I_{D,i,i-1}$ *iff* $\Omega_{Datalog}(\Omega(s_i))$ *is satisfied by* D.

The proof of this claim is also basically by definition, in this case using the definition of $\Omega_{Datalog}(\Omega(s_i))$.

Let (s_{i-1}, ψ, s_i) be the transition from the $i{-}1^{st}$ state and the i^{th} state of A. The fourth and last claim of this subsubsection is concerned with the relationship between the formula ψ^- and $\Omega_{Datalog}(\psi^-)$. We recall that the formula ψ^- is a Boolean combination of negated positive sentences over $\mathrm{Sch}_{\mathrm{Acc}}$ and these sentences do not have any atom of the form $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{x})$. The sentence $\Omega_{Datalog}(\psi^-)$ is obtained from ψ^- 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 $R_{\text{post}}(\bar{x})$ by

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

CLAIM 4. *Let:*
— Sch *be a schema,*
— A *be a linear A-automaton,*
— $d = (s_{i-1}, \psi, s_i)$ *be the transition of* A *from the state* s_{i-1} *to the state* s_i*,*
— $(\text{AcM}, \bar{b})$ *be the access associated with the transition* d*,*
— D *be a database over* Sch_{ext}
Then ψ^- *is satisfied by* $(I_{D,i-1,i-2}, (\text{AcM}, \bar{b}), I_{D,i-1,i-1})$ *iff* $\Omega_{Datalog}(\psi^-)$ *is satisfied by* D*.*

The proof of this claim follows from the definition of $\Omega_{Datalog}(\psi^-)$.

4.5.2. Decomposition of the Datalog program $\mathcal{P}_A$. The next claim shows that $\mathcal{P}$ can be decomposed into subprograms $\mathcal{P}_0, \mathcal{P}_{0,1}, \mathcal{P}_1, \mathcal{P}_{1,2}, \mathcal{P}_2 \ldots, \mathcal{P}_h$, $h = \text{height}(A)$, that can be evaluated "in sequence": (i) the program $\mathcal{P}_0$ contains the rules related to predicate $\text{Answer}(\text{AcM})_j$ and $\text{IntAnswer}(\text{AcM})_j$ for any access method AcM and any integer j less or equal to h; (ii) letting i be an integer between 1 and h, the Datalog program $\mathcal{P}_{i-1,i}$ contains all rules of $\mathcal{P}_A$ related with the transition from s_{i-1} to s_i, in particular the rules related to ReachC_i and the Datalog program $\mathcal{P}_i$ contains all the rules of $\mathcal{P}_A$ related with the transitions from s_i to itself in A.

CLAIM 5. $\mathcal{P}_A^{full}(D) = \mathcal{P}_h^{full}(\mathcal{P}_{h-1,h}(\mathcal{P}_{h-1}^{full}(\ldots \mathcal{P}_1^{full}(\mathcal{P}_0^{full}(D))\ldots)))$ *for every database* D *over the extensional schema of* $\mathcal{P}_A$*.*

PROOF OF CLAIM 5. It is obvious that the right hand side is included in the left hand side. The other direction follows from the fact that if $\text{head} :- \text{body}$ is a rule in $\mathcal{P}_i$ then body contains only atoms that appear as heads of rules in $\mathcal{P}_0 \cup \mathcal{P}_1 \cup \ldots \cup \mathcal{P}_i$. □

4.5.3. Construction of the partial LTrans *paths associated with the partial evaluations of $\mathcal{P}_A$.*
First, we need the following notation:
— D_i is the instance equal to $\mathcal{P}_i^{full}(\mathcal{P}_{i-1,i}^{full} \cdots \mathcal{P}_0^{full}(D))$,
— $D_{i-1,i}$ is the instance equal to $\mathcal{P}_i^{full}(\mathcal{P}_{i-1,i}^{full} \cdots \mathcal{P}_0^{full}(D))$,
— $J_{\rightarrow s_i}(D)$ is the instance over Sch such that
  — if $i = 1$ then each relation T is interpreted as in the relation $D(\text{IntBackgroundT}_0)$,
  — Otherwise, let AcM and $\bar{b}$ be the access method and the binding associated with the formula ψ in the transition (s_{i-1}, ψ, s_i) following Condition (Deterministic-Transition),
    — for each relation name T different from $\text{Rel}(\text{AcM})$, the tuples in $J_{\rightarrow s_i}(D)(T)$ are those in $\text{Sch}(D_{i-1})(T)$,
    — the tuples in $J_{\rightarrow s_i}(D)(\text{Rel}(\text{AcM}))$ are those in the union of $\text{Sch}(D_{i-1})(\text{Rel}(\text{AcM}))$ along with those in the maximal well-formed output for the access $(\text{AcM}, \bar{b})$ on a instance I where $I(\text{Rel}(\text{AcM}))$ is interpreted as in $D(\text{IntBackground}(\text{Rel}(\text{AcM}))_{i-1})$
— $\mathcal{P}_i^{full,j}(D)$ is the result of the first j iterations of the rules of $\mathcal{P}_i$ over D,
— D_i^j is the instance equal to $\mathcal{P}_i^{full,j}(D_{i-1,i})$
Thus D_i is the result of performing the first i stages of the Datalog program until they reach a fixed point, $J_{\rightarrow s_i}(D)$ is the instance obtained on the Datalog schema corresponding to entering state s_i, D_i^j is the result of performing the first $i-1$ stages completely followed by j iterations of the i^{th} stage of the Datalog program. We consider in this paper a non-deterministic semantics of Datalog evaluation. At each iteration, only

one new fact is added. Recall that $\mathrm{Sch}(D')$ for any D' above is the translation of these to the original schema.

The construction of the LTrans path accepted by A is done in several steps. The first step, Lemma 4.15, concerns creating the transitions that jump between components. It shows that for each $i > 1$, if $D_i(\mathrm{ReachC}_i)$ is true then there exists a LTrans transition t from the instance $\mathrm{Sch}(D_{i-1})$ to the instance $J_{\to s_i}(D)$, such that t satisfies the formula of the transition of A going from the $i-1^{st}$ state to the i^{th} state. The second lemma, Lemma 4.16, shows that when the Datalog program adds certain tuples, then there is a corresponding path consisting of self-loops in the automaton that adds these tuples. Formally, it shows that for each $i \geq 1$ if $D_i(\mathrm{ReachC}_i)$ is true and $\mathrm{Sch}(D_i) - J_{\to s_i}(D)$ is true, then there exists a LTrans path ρ from $J_{\to s_i}(D)$ to $\mathrm{Sch}(D_i)$ and a run r on ρ from s_i to itself. The subpaths and subruns obtained from the two lemmas are concatenated to obtain a path accepted by A.

We now prove our first lemma about mimicking Datalog instances by schema instances that are reachable via runs of the automaton.

LEMMA 4.15. *Let:*
— Sch *be a schema,*
— $A = (S, s_1, F, \delta)$ *be a linear A-automaton,*
— $\mathcal{P}_A$ *and* $\mathcal{P}'_A$ *be the Datalog program and the positive sentence obtained from* A*,*
— D *be an instance over* $\mathrm{Sch}_{\mathrm{ext}}$*.* D *does satisfy* $\mathcal{P}_A$ *and* D *does not satisfy* $\mathcal{P}'_A$*,*
— i *be an integer strictly greater than* 1*,*
— $d = (s_{i-1}, \psi, s_i)$ *be the transition of* A *from the* $i-1^{st}$ *state to the* i^{th} *state*
If $D_i(\mathrm{ReachC}_i)$ *is true then there exists a* LTrans *transition* t *from the instance* $\mathrm{Sch}(D_{i-1})$ *to the instance* $J_{\to s_i}(D)$ *and this transition satisfies* ψ*.*

PROOF OF LEMMA 4.15. We assume that $D_i(\mathrm{ReachC}_i)$ is true. Let $(\mathrm{AcM}, \bar{b})$ be the access associated with d following Condition (Deterministic-Transition). Let t be the LTrans transition equal to $(\mathrm{Sch}(D_{i-1}), (\mathrm{AcM}, \bar{b}), J_{\to s_i}(D))$: one can easily see that this is an LTrans transition. The lemma is proven in two steps: first we show that t satisfies ψ^+, then we prove that t satisfies ψ^-.

Satisfaction of ψ^+. We recall that $\mathrm{IsValidBinding}_d$ is the formula built from ψ^+ in the construction of $\mathcal{P}_A$, obtained from ψ^+ as follows:
— The atom $\mathrm{IsBind}_{\mathrm{AcM}}(\bar{x})$ is replaced by $\mathrm{Bind}_{\mathrm{AcM}}(\bar{b})$.
— Each atom $R_{\mathrm{pre}}(\bar{y})$ is replaced by $\bigvee_{j<i} \mathrm{ViewR}_j(\bar{y})$.
— Each atom $R_{\mathrm{post}}(\bar{y})$ is replaced by $\mathrm{IntAnswer}(\mathrm{AcM})_{i-1}(\bar{b}, \bar{y}) \vee \bigvee_{j<i} \mathrm{ViewR}_j(\bar{y})$.

Let AcM be the access method associated with the transition d. We define by K_D the following instance:
— for each relation T which is different than $\mathrm{Rel}(\mathrm{AcM})$, $K_D(R)$ contains the tuples in $\bigcup_{j<i} D_{i-1}(\mathrm{ViewT}_j)$,
— for the relation R which is equal to $\mathrm{Rel}(\mathrm{AcM})$, $K_D(R)$ contains the tuples in $\bigcup_{j<i} D_{i-1}(\mathrm{ViewR}_j) \cup D_0(\mathrm{IntAnswer}(\mathrm{AcM})_{i-1})$.

Because $D_i(\mathrm{ReachC}_i)$ is true, $\mathrm{IsValidBinding}_d$ is satisfied by K_D. Also recall that $J_{\to s_i}(D)$ is the instance obtained from $\mathrm{Sch}(D_{i-1})$ by performing the access $(\mathrm{AcM}, \bar{b})$ in any instance I' such that $I'(\mathrm{Rel}(\mathrm{AcM}))$ is interpreted as in $D(\mathrm{IntBackground}(\mathrm{Rel}(\mathrm{AcM}))_{i-1})$. By Claim 2, ψ^+ is satisfied by $(\mathrm{Sch}(D_{i-1}), (\mathrm{AcM}, \bar{b}), J_{\to s_i}(D))$.

Satisfaction of ψ^-. Because D does not satisfy $\mathcal{P}_{A'}$, D satisfies $\Omega_{Datalog}(\psi^-)$. Due to Claim 4, ψ^- is not satisfied by $(I_{D,i-1,i-2}, (\mathrm{AcM}, \bar{b}), I_{D,i-1,i-1})$. We deduce that $J_{\to s_i}(D)$ is included in $I_{D,i-1,i-1}$. Thus by monotonicity of ψ^-, t satisfies ψ^-.

We can conclude that t satisfies ψ. □

We denote by Access_i^j the set of accesses $(\text{AcM}, \bar{b})$ such that $\bar{b}$ is in $D_i^j(\text{Done-Access}_{\text{AcM}}^i(\bar{b}))$. We can associate a total order in these accesses such that $(\text{AcM}, \bar{b})$ is less than $(\text{AcM}', \bar{b}')$ iff for each $k \leq j$, if $(\text{AcM}, \bar{b})$ is in Access_i^k then $(\text{AcM}', \bar{b}')$ is in Access_i^k.

The next lemma explains how to build a LTrans path that adds the same tuples that appeared in the ViewR_i relations during the j^{th} iteration of $\mathcal{P}_i$.

In the following lemma, we use the notation $\mathcal{P}_i^0(D)$ which is equal to D.

LEMMA 4.16. *Let*
— Sch *be a schema,*
— LTrans *be the LTS associated with* Sch*,*
— A *be a linear* A*-automaton,*
— $\mathcal{P}_A$ *and* $\mathcal{P}'_A$ *be the Datalog program associated with* A *and the positive sentence associated with* A*,*
— i *and* j *be two integers such that* $1 \leq i \leq h$ *and* $0 \leq j$*,*
— D *be an instance over the extensional schema of* $\mathcal{P}_A$ *such that* D *satisfies* $\mathcal{P}_A$ *and does not satisfy* $\mathcal{P}'_A$

If there exists an access method AcM *such that* $D_j^i(\text{Done-Access}_{\text{AcM}}^i)$ *is not empty then there exists an* LTrans *path* ρ *such that*

(*1*) *the initial instance is* $J_{\rightarrow s_i}(D)$*,*
(*2*) *the accesses appearing in* ρ *are the accesses in* Access_i^j*., and the order defined in* Access_i^j *is the same as the order defined by* ρ*,*
(*3*) *the final instance of* ρ *denoted by* J_ρ *satisfies that for each access* $(\text{AcM}, \bar{b})$ *in* Access_i^j*,* $(\text{AcM}, \bar{b}, J_\rho)$ *is equal to* $(\text{AcM}, \bar{b}, I_{D,i,i-1})$. $\text{Sch}(D_i^j)$ *is included in* J_ρ*,*
(*4*) *there exists a run from* s_i *to* s_i *on* ρ*.*

Otherwise ρ *is empty and we define* J_ρ *by* $J_{\rightarrow s_i(D)}$.

PROOF OF LEMMA 4.16. We proceed by an induction on on the number of iterations j of the subprogram $\mathcal{P}_i$ that have been applied.

Base Case $j = 0$.
Access_i^0 is empty. Thus ρ is empty and J_ρ is equal to $J_{\rightarrow s_i(D)}$.

Inductive case $j \geq 1$.
Let ρ be the path built for $j - 1$ by using the induction hypotheses. Let $\bar{v}$ be the tuple added at the j^{th} iteration of $\mathcal{P}_i$. We notice that there are two main cases: either $\bar{v}$ is added to a relation ViewR_i or $\bar{v}$ is added to a relation $\text{Done-Access}_{\text{AcM}}^i$.

Case 1: Tuple $\bar{v}$ *is added to* ViewR_i.
We consider two subcases following if ρ is empty or not.

— If ρ is empty. Then all the relations $\text{Done-Access}_{\text{AcM}}^i$ are empty. Due to the construction of the body formula φ of the rule associated with ViewR_i, φ cannot be satisfied by D_j and thus the tuple $\bar{v}$ cannot be added to ViewR_i. Therefore, we have a contradiction of the inductive hypotheses.
— Suppose now that ρ is not empty. Due to the construction of the body formula associated with the rule associated with ViewR_i, there exists an access $(\text{AcM}, \bar{b})$ such that $\bar{v}$ belongs to $(\text{AcM}, \bar{b}, I_{D,i,i-1})$ and $\bar{b}$ belongs to $D_i^{j-1}(\text{Done-Access}_{\text{AcM}}^i)$. By the induction hypothesis on ρ, $\bar{v}$ belongs to $I_\rho(R)$. Therefore, ρ still satisfies the inductive invariant for j.

Case 2: tuple $\bar{v}$ is added to $\text{Done-Access}^{i}_{\text{AcM}}$.

The goal is to extend the path ρ. Because $\bar{v}$ is added to $\text{Done-Access}^{i}_{\text{AcM}}$ there exists a transition d of the form (s_i, ψ, s_i) such that

— ψ^+ is equal to $\exists \bar{x}\, \text{IsBind}_{\text{AcM}}(\bar{x}) \wedge \psi_1$ following Condition (Deterministic-Access-Method) of linear automata.
— ValidBind_d is satisfied by D_i^{j-1}.

We construct ρ' by addying to ρ the transition $t = (J_\rho, \text{AcM}, \bar{b}, J_\rho + (\text{AcM}, \bar{b}, I_{D,i,i-1}))$. We now have to prove that t satisfies ψ.

Satisfaction of $\Omega(s_i)$. We first prove that $J_\rho + (\text{AcM}, \bar{b}, I_{D,i,i-1})$ satisfies $\Omega(s_i)$. We recall that $\Omega(s_i)$ is a Boolean combination of negated positive sentences over Sch associated with s_i due to Condition (Negative-Invariant). The sentence $\Omega_{Datalog}(\Omega(s_i))$ is obtained from $\Omega(s_i)$ by replacing each atom $R(\bar{x})$ by

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

Because D does not satisfy $\mathcal{P}'_A$, D satisfies $\Omega_{Datalog}(\Omega(s_i))$. Due to Claim 3, $I_{D,i,i-1}$ satisfies $\Omega(s_i)$. Due to Property 3 of Lemma 4.16, $I_(\rho) + (\text{AcM}, \bar{b}, I_{D,i,i-1})$ is included in $I_{D,i,i-1}$. Because $\Omega(s_i)$ is monotone, $J_\rho + (\text{AcM}, \bar{b}, I_{D,i,i-1})$ satisfies $\Omega(s_i)$.

Satisfaction of ψ^+. Due to Claim 1, the transition $(\text{Sch}(D_i^{j-1}), (\text{AcM}, \bar{b}), \text{Sch}(D^{j-1}) + (\text{AcM}, \bar{b}, I_{D,i,i-1}))$ satisfies ψ^+. Because $\text{Sch}(D_i^{j-1})$ is included in J_ρ and ψ^+ is monotone, t satisfies ψ^+.

Satisfaction of ψ^-. First, we have proven above (in the paragraph headed by "Satisfaction of $\Omega(s_i)$") that $J_\rho + (\text{AcM}, \bar{b}, I_{D,i,i-1})$ satisfies $\Omega(s_i)$. We recall Property (NegativeInv +) of linear A-automata A (see the text after Definition 4.11): For each transition (s, ψ, s) if any instance I satisfies $\Omega(s)$ then for any LTrans transition $t = (I', (\text{AcM}, \bar{b}), I'')$, if $I' \subseteq I'' \subseteq I$ then t satisfies ψ^-. Applying Property (NegativeInv +) of linear A-automata, t satisfies ψ^-.

Because t satisfies ψ^- and ψ^+, t satisfies ψ. Therefore $r.d$ is a run to s_i on $\rho.t$.

This finishes the induction. □

4.5.4. Construction of the accepted path. Let D be an instance satisfying $\mathcal{P}_A$ and not satisfying $\mathcal{P}'_A$. We now build the path ρ and the accepting run r on ρ showing that A is not empty. First, we recall that for each $i > 1$, the rule of ReachC_i has in its body formula the atom ReachC_{i-1}. Because D satisfies $\mathcal{P}_A$, $\mathcal{P}_A^{full}(D)(\text{ReachC}_h)$ is true. Therefore for each i, $D_{i-1,i}(\text{ReachC}_i)$ holds. Due to Lemma 4.15, for each $i > 1$, there exists an LTrans transition $t_{i-1,i}$ from the instance D_{i-1} to $J_{\to s_i}(D)$ and the transition (s_{i-1}, ψ_i, s_i) is a run on $t_{i-1,i}$. For each $i \geq 1$, a LTrans path ρ_i and a run r_i on ρ_i are built as follows:
— if $J_{\to s_i}(D)$ is equal to $\text{Sch}(D_i)$ then ρ_i and r_i are empty;
— otherwise let j be an integer such that D_i^j is equal to D_i. Let ρ_i from $J_{\to s_i}(D)$ to D_i and r_i from s_i to itself be the path and run obtained using Lemma 4.16.

We define the LTrans path $\rho = \rho_1 \cdot t_{1,2} \cdots \rho_{h-1} \cdot t_{h-1,h}$ and the run $r = r_1 \cdot (s_1, \psi_2, s_2) \cdots r_{h-1} \cdot (s_{h-1}, \psi_h, s_h)$. Then r is an accepting run on ρ, and therefore A is not empty.

4.6. From the emptiness of the linear automaton to the containment of the Datalog program in positive query.

We now show that if A has an accepting run, then there is a database D of Sch_{ext} that is a model of $\mathcal{P}_A$ but not of $\mathcal{P}'_A$. Towards this goal, let $\rho = t_1, \ldots, t_n$ with $t_j =$

$(\mathrm{I}_l, (\mathrm{AcM}_l, \bar{b}_{j_k}, \mathrm{I}_{l+1})$ be a path on which A has an accepting run $r = d_1, \ldots, d_n$ with $d_l = (s_{i_l}, \varphi_{i_l}, s_{i_{l+1}}) \in \delta$ for $1 \leq l \leq n$.

We define a set of instances K_i and J_i and H_i over Sch. These instances are used to build D. The instance K_0 is equal to I_1. Let i be an integer less or equal to n, and $d_{m(i)}$ be the transition in r going from the state s_i to the state s_{i+1}. Let $t_{m(i)} = (I_{m(i)}, (\mathrm{AcM}_{m(i)}, \bar{b}_{m(i)}), I_{m(i)+1})$ be the LTrans transition of ρ associated with $d_{m(i)}$. We define:

— K_i to be $I_{m(i)+1} - I_{m(i)}$,
— H_i to be $I_{m(i)}$,
— if $i = 1$ then $J_1 = H_1 - K_0$ otherwise $J_i = H_i - (H_{i-1} \cup K_{i-1})$.

We build the database D over $\mathrm{Sch}_{\mathrm{ext}}$ as follows: for each $1 \leq i \leq h$ and each relation $R \in \mathrm{Sch}$, $D(\mathrm{BackgroundR}_i)$ is interpreted as in $J_i(R)$ and $D(\mathrm{IntBackgroundR}_{i-1})$ is interpreted as in $K_{i-1}(R)$. For each AcM, the tuples in $\mathrm{Bind}_{\mathrm{AcM}}(D)$ are the tuples $\bar{b}$ such that $(\mathrm{AcM}, \bar{b})$ appearing in ρ.

We first show that D is a model of $\mathcal{P}_A$. Let $\mathcal{P}_A^{full}(D)$ be the fixed point of applying $\mathcal{P}_A$ to D, and we claim that for all $i \leq h$, where h is the height of A, for each relation name R

$$H_i(R) \subseteq \bigcup_{j \leq i} \mathcal{P}_A^{full}(D)(\mathrm{ViewR}_i)$$

This can be shown easily by induction on the length of the run r. Note that by the definitions of $\mathcal{P}_A$ and D it holds that

$$H_i(R) \supseteq \bigcup_{j \leq i} \mathcal{P}_A^{full}(D)(\mathrm{ViewR}_i)$$

It follows that D satisfies $\mathcal{P}_A$. We prove now that D does not satisfy $\mathcal{P}'_A$.

We recall Property (NegativeInv †) of a linear A-automaton: For each transition (s, ψ, s') if a LTrans transition $(I, (\mathrm{AcM}, \bar{b}), I')$ satisfies ψ then I' satisfies $\Omega(s)$.

For each $i \leq h$,

— By construction of D, H_i is equal to $D_{i,i-1}$ and $H_i \cup K_i$ is equal to $D_{i,i}$.
— Due to the Property (NegativeInv †) of linear A-automaton, H_i satisfies $\Omega(s_i)$. Due to Claim 3, $\Omega_{Datalog}(\Omega(s_i))$ is satisfied by D.
— $(H_i, (\mathrm{AcM}, \bar{b}), H_i \cup K_i)$ satisfies ψ_i^- where (s_i, ψ_i, s_{i+1}) is a transition of A and $(\mathrm{AcM}, \bar{b})$ is the access associated with ψ_i following Condition (Deterministic-Transition) . Due to Claim 4, $\Omega_{Datalog}(\psi_i^-)$ is satisfied by D.

Because $\mathcal{P}'_A$ is the dual of the conjunction of the sentences $\Omega_{Datalog}(\Omega(s_i))$ and $\Omega_{Datalog}(\psi_i^-)$ for $i \leq h$, D does not satisfy $\mathcal{P}'_A$. □

4.7. Containment of Datalog in positive queries, and completion of the proof of the main A-automaton theorem

Let us review what we have accomplished thus far: we have reduced questions about our logic to non-emptiness of the automata, and we have reduced checking non-emptiness of an automaton to determining whether a Datalog program is contained in a positive query. To complete the proof of Theorem 4.9 we need the following new result, that generalizes a theorem of Chaudhuri and Vardi [Chaudhuri and Vardi 1997a]:

PROPOSITION 4.17. *The containment problem of a Datalog program P in a positive first-order sentence φ, where both P and φ may contain constants, is in 2EXPTIME.*

Theorem 4.9 follows from the proposition and the reduction given earlier.

PROOF. We first show the argument in the absence of constants and equality atoms. We adapt the proof that containment of a Datalog program P in a union of conjunctive

queries can be decided in 2EXPTIME [Chaudhuri and Vardi 1997b]. 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 automaton, A_P, of size exponential in the size of P. In the same way, each θ_i is associated with a non-deterministic tree automaton, A_{θ_i}, which is of size exponential in P and θ_i. Theorem 5.11 of [Chaudhuri and Vardi 1997b] 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 [Chaudhuri and Vardi 1997b]. It is known that for any non-deterministic tree automaton A and A', there exists an automaton A'' such that A'' is equivalent to the union of A and A' and $|A''| \in O(|A| + |A'|)$ [Comon et al. 1997]. We can deduce that there exists a non-deterministic tree automaton A_φ of size exponential in φ equivalent to $\bigcup_i A_{\theta'_i}$. 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 [Chaudhuri and Vardi 1997b] 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 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 that the right-hand side is an exponential disjunction of conjunctive queries, and then proceed as in the case without constants above. □

4.8. Hardness of A-automata non-emptiness

We recall that we promised to use the connection of A-automata to Datalog to show a hardness result for A-automata. Specifically, earlier in this section we stated Theorem 4.10 without proof:

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 [Chaudhuri and Vardi 1997a].

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 automaton 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 $(s_0, B_r^{pre}(), s_f) \in \delta$.

For any query Q over Sch, $Q^{\mathbf{pre}}$ is the query obtained by changing R to $R_{\mathbf{pre}}$.

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 intentional 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 intentional 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 $f_1 \ldots f_n$ 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.

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 intentional 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. $\rho = \varnothing$, 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 intentional predicates are unchanged, but by monotonicity of the Datalog program P, the values of the intentional 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 intentional predicates is a subrelation of the intentional predicates calculated by the Datalog program P. In addition, since an accepting 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 intentional 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. □

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.

5. TRADING OFF EXPRESSIVENESS FOR COMPLEXITY OF ANALYSIS

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 [Clarke et al. 2000].

For a relational schema Sch, we define the vocabulary $\mathrm{Sch}_{0-\mathrm{Acc}}$ as in $\mathrm{Sch}_{\mathrm{Acc}}$ but instead of the 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 2.3 and 2.4 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 use this restricted logic to express the access order restrictions of Example 2.4. We now see that by curtailing the expressiveness, the complexity goes down significantly.

THEOREM 5.1. *Satisfiability of an* $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ *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 [Emerson 1990]. 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$. 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 5.2. *An* $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ *formula* φ *is satisfiable iff there exists a path* ρ *which satisfies* φ *and which has 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 *and* Sch*.*

PROOF. The if-part is trivial, so we turn to the only-if. 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 $\mathrm{AcM} \in \mathrm{Sch}$ 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 number of bindings used is bounded by a polynomial function in the sizes of φ and the schema. □

Given the lemma, we can now apply the following algorithm which is easily seen to be in NPSPACE:

(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 5.2). 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ 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 $s = \mathrm{I}_1 \dots \mathrm{I}_n$ 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 that match the two forms above is denoted $T(I, B)$. For each i, we denote by (1) $T(i)$ the set of transitions of the form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$. (2) $t_{i,\rightarrow}$ the transition $(I_i, A(i), I_{i+1})$. (3) $P(i)$ the set of propositions associated with the transitions of the form $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$. (4) $p_{i,\rightarrow}$ the proposition associated with the transition (I_i, A_i, I_{i+1}). The set of all such propositions is denoted Σ. The formula $\overline{\varphi}$ that we will present will define words over alphabet 2^{Σ}. Intuitively, each letter of a word is 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:

$$\bigwedge_{i, p \in P(i)} \mathsf{G}\Big(p \Rightarrow \big(\bigvee_{p' \in P(i)} p' \,\mathsf{U}\, p_{i,\rightarrow}\big)\Big) \wedge \bigwedge_{i} \Big(p_{i,\rightarrow} \Rightarrow \mathsf{X}\big(\bigvee_{p'' \in P(i+1)} p'' \vee p_{i+1,\rightarrow}\big)\Big) \wedge \big(\bigvee_{p''' \in P(0)} p''' \vee p_{0,\rightarrow}\big)$$

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.

Extension of the Proof of Theorem 5.1 for exact and idempotent paths. 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 of Theorem 5.1, and we explain the modifications needed for idempotent and exact accesses. We first claim that Lemma 5.2 still holds with these conditions. For idempotent accesses, we can use the same process as in the proof of Lemma 5.2: 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 5.2 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 5.2, the rest of the argument proceeds via rewriting as above.

5.1. 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 2.4. It is easy to see that Q is LTR over all accesses iff it is LTR over access paths of size $|Q|$ – a counter example 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 5.3. *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.*

PROOF. We start out with the proof for general accesses.

Hardness For brevity, we explain the argument only when an initial instance for paths can be fixed, leaving the general proof for the reader. Given a boolean combination σ of positive existential first-order sentences, along with S a "schema" restricting positions $R[i]$ of relation R to take values only in some finite set $Type(R[i])$. We can efficiently create an initial instance I_0 and an $\mathrm{AccLTL}(X)(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$ sentence φ such that there is a path from I_0 satisfying φ exactly when there is a database instance conforming to S and satisfying σ: we let the signature for φ contain the predicates of σ, along with unary predicates for $R[i]$, where the predicates of σ have free access while those for $R[i]$ have no access. We let I_0 contain $Type(R[i])$ as the values for predicate $R[i]$. φ will simply use next operators to assert that the given path consists of a sequence of accesses to each predicate of σ in turn, reaching a position satisfying a final condition consisting of σ conjoined with predicates $R[i](x)$ whenever x occurs in position i of relation R within σ. Thus φ asserts that after all of the relations are revealed, σ holds. It is easy to verify that this reduction is as required.

In particular, non-containment of positive relational algebra queries in which positions of particular attributes can be restricted to have "finite type" – to take values in some finite set – can be reduced to the complement of the satisfiability 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 make use of the "boundedness lemma", Lemma 5.2, 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 φ and which further has 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 size of φ.

We will adapt the technique used in the proof of Theorem 5.1.

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. We denote by n the length of $\bar{\mathrm{I}}$, minus one. 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 5.2).

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

— $(I_i, (\mathrm{AcM}, \vec{b}), I_i)$ where $\vec{b}$ is in B and compatible with AcM.

— (I_i, A_i, I_{i+1}), where A_i is any access.

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 $T(0)^*, t_{0,\rightarrow}, \ldots, t_{n-1,\rightarrow} T(n)^*$. The translation from φ to ψ can be obtained in polynomial time. The main difference from the proof of Theorem 5.1 is that we need to express the "sanity axioms" using only the operator X, whereas in the proof of Theorem 5.1, 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 5.2 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 5.2 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. Thus the previous arguments are still correct in this context. We can conclude that these results can be extended for formulas with inequalities. □

6. EXTENSIONS AND LIMITS

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

6.1. 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 2.5.

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 6.1. *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 5.1)*

— *satisfiability of* $\mathrm{AccLTL}(\mathsf{X})(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$ *is in* Σ^P_2 *(hence* Σ^P_2*-complete by Theorem 5.3)*

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 6.2. *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 and inclusion dependencies for relational databases to the problem of the unsatisfiability of a AccLTL^+ with inequalities formula. For simplicity we again assume all positions carry the same type and verify the reduction over instances where all relations are nonempty. This proof uses the same kind of construction as in the proof of Theorem 3.1. The presence of inequalities allows us to simplify the check functional dependencies.

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. As in the prior proof, we will have a relation $\mathrm{Succ}(R)$ of arity $2k$ representing a successor relation on tuples of R. 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 describe the formula ψ' such that ψ' is satisfiable iff σ is not implied by Γ.

$$\varphi'_{\mathrm{order}} \wedge \varphi'_{\mathrm{Beg}(R)} \wedge \mathsf{X}(\varphi'_{\mathrm{next(R)}} \mathsf{U} (\varphi'_{\mathrm{End}(R)} \wedge \mathsf{X}\varphi'_{\mathrm{verify}}))$$

First, $\varphi'_{\mathrm{order}}$ verifies 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 easily done. For example, for the first primary key condition, we

add for each $i \leq k$ a conjunct:

$$\mathsf{G}\big(\neg \exists \vec{s}\, \vec{t}\, \vec{u}\ \mathrm{Succ}(R)(\vec{s},\vec{t}) \wedge \mathrm{Succ}(R)(\vec{s},\vec{u}) \wedge t_i \neq u_i\big)$$

Similarly to the proof of Theorem 3.1, we can enforce that at any point relation $\mathrm{Beg}(R)$ has 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 further details on 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. $\varphi'_{\mathrm{verify}}$ will be of the form

$$\varphi'_{\mathrm{CheckFds}} \wedge \mathsf{X}(\varphi_{\mathrm{CheckIds}} \wedge \mathsf{X}\varphi'_{\mathrm{CheckFdFailure}})$$

where $\varphi_{\mathrm{CheckIds}}$ is the same subformula as defined in Theorem 3.1 but $\varphi'_{\mathrm{CheckIds}}$ and $\varphi'_{CheckFdFailure}$ are different from Theorem 3.1 as they cannot use negation.

Previously in this proof, we have shown how to express that a primary key is satisfied by the relation$\mathrm{Succ}(R)$. This construction can be generalized to express the satisfactions of the functional dependencies in Γ given the subformula $\varphi'_{kwCheckIds}$. Finally, a variation of the construction, $\varphi'_{\mathrm{CheckFdFailure}}$ checks the failure of the distinguished functional dependency σ.

We will focus on the formulas enforcing an inclusion dependency id from relation R to relation S, assuming for simplicity that the dependency requires that for each tuple u in R there is a tuple v in S such that u and v agree on the first k positions. The formula enforcing this is of the form:

$$\begin{aligned}
&\exists \vec{t}\ \mathrm{Beg}(R)_{\mathbf{pre}}(\vec{t}) \wedge \mathrm{IsBind}_{\mathrm{CheckIncDep}(id)}(\vec{t}) \wedge \mathrm{CheckIncDep}(id)_{\mathbf{post}}(id)(\vec{t})\\
&\wedge \exists \vec{v}\ \exists \vec{w}(\mathrm{Succ}(S)_{\mathbf{pre}}(\vec{v},\vec{w}) \vee \mathrm{Succ}(S)_{\mathbf{pre}}(\vec{w},\vec{v})) \wedge \bigwedge_{i\leq k} t_i = v_i)\\
&\mathsf{X}[(\exists \vec{t}\, \vec{u}\ \mathrm{Succ}_{\mathbf{pre}}(R)(\vec{t},\vec{u}) \wedge \mathrm{CheckIncDep}(id)_{\mathbf{pre}}(\vec{t}) \wedge\\
&\mathrm{IsBind}_{\mathrm{CheckIncDep}(id)}(\vec{u}) \wedge \mathrm{CheckIncDep}(id)_{\mathbf{post}}(id)(\vec{u})\\
&\wedge \exists \vec{v}\ \exists \vec{w}(\mathrm{Succ}(S)_{\mathbf{pre}}(\vec{v},\vec{w}) \vee \mathrm{Succ}(S)_{\mathbf{pre}}(\vec{w},\vec{v})) \wedge \bigwedge_{i\leq k} u_i = v_i)\\
&\mathsf{U}\,(\exists \vec{w}\ \mathrm{End}(R)_{\mathbf{pre}}(\vec{w}) \wedge \mathrm{CheckIncDep}(id)_{\mathbf{pre}}(\vec{w}))]
\end{aligned}$$

This describes an access path p that begins by using a tuple that is in $\mathrm{Beg}(R)$, performing an access to $\mathrm{CheckIncDep}(id)$ with this tuple, checking that this tuple has a witness for id and adding it to $\mathrm{CheckIncDep}(i)$. The formula states that p 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. p will only stop when the last tuple in the order is in $\mathrm{CheckIncDep}(id)$.

We claim, as in Theorem 3.1, that if this sentence is satisfied by path p, the instance $J(p)$ will witness failure of the implication of σ from Γ, where $J(p)$ is obtained from the final instance in p by taking tuples in the first projection of $\mathrm{Succ}(R)$ as well as those in $\mathrm{End}(R)$.

Conversely if I witnesses the failure of the implication of σ by Γ, we can form a witness $p(I)$ to the sentence as in Theorem 3.1.

□

The reader may want to look at Figure 4 for a view to how the languages with inequalities relate to the languages defined previously.

6.2. 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 $S_{\text{pre}}, S_{\text{post}}$ of each relation symbol S and the proposition $\text{IsBind}_{\text{AcM}}$.

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 $\text{CTL}_{EX}(L)$, but instead of CTL we might as easily have said "basic modal logic" or Hennessy-Milner Logic [Emerson 1990], since we only need the power of the most basic existential modality to get undecidability. $\text{CTL}_{EX}(L)$ 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 $(S, t) \vDash \varphi$, where t is a transition $(\text{I}, \text{AC}, \text{I}')$ 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, $(S, t) \vDash \mathsf{EX}\varphi$ iff there is a successor t' of t such that $(S, t') \vDash \varphi$. Note that Deutsch et. al. [Deutsch et al. 2007] 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 [Deutsch et al. 2007]) allow one to describe properties of the input (analogous to our larger signature Sch_{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 6.3. *Satisfiability of* $\text{CTL}_{EX}(\text{FO}^{\exists+}_{0-\text{Acc}})$ *is undecidable.*

PROOF. We reduce from the problem of implication of a functional dependency from a set of functional dependencies and inclusion dependencies for relational databases. This is known to be undecidable [Chandra and Vardi 1985].

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 $(\text{Fill}_R, \varnothing)$ returns an essentially random configuration of R. We also have additional relations $\text{Chk}^{\text{FD}}(R)$, having twice the arity of R and $\text{CheckIncDep}(R)$ 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:

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

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 $\varphi_{\neg\sigma}$:

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

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

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

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 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. □

Language	Complexity	DjC	FD	DF	AccOr
$\mathrm{AccLTL}(\mathrm{FO}^{\exists+,\neq}_{\mathrm{Acc}})$	undecidable (Thm 3.1)	Yes	Yes	Yes	Yes
$\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$	undecidable (Thm 3.1)	Yes	No	Yes	Yes
AccLTL^{+}	in 3EXPTIME (Thm 4.2)	Yes	No	Yes	Yes
A-automata	2EXPTIME-compl. (Thms 4.9 & 4.10)	Yes	No	Yes	Yes
$\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{0-\mathrm{Acc}})$	PSPACE-compl. (Thm 5.1)	Yes	No	No	Yes
$\mathrm{AccLTL}(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$	PSPACE-compl. (Thm 6.1)	Yes	Yes	No	Yes
$\mathrm{AccLTL}(\mathsf{X})(\mathrm{FO}^{\exists+,\neq}_{0-\mathrm{Acc}})$	Σ^P_2-compl. (Thm 6.1)	Yes	Yes	No	No

Fig. 3. Complexity and application examples for path specifications.

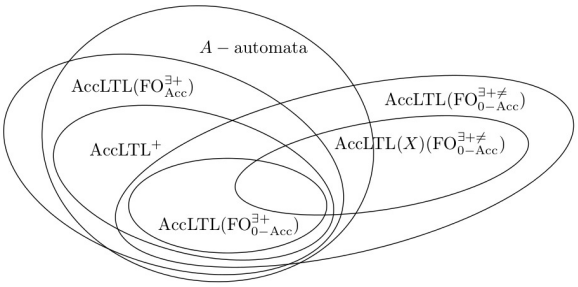

Fig. 4. Inclusions between language classes.

7. 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 4 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 $\mathrm{AccLTL}(\mathrm{FO}^{\exists+}_{\mathrm{Acc}})$, can not do.

Table 3 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 $\mathrm{FD}, \mathrm{DF}, \mathrm{AccOr}$ refer to functional dependencies, dataflow restrictions, and access order restrictions, respectively.

Our work leaves open a number of questions concerning the logics we study. We believe that examination of the translation to A-automata, along with a finer analysis of A-automata emptiness will show that satisfiability of AccLTL^{+} is 2EXPTIME-complete; but the results presented here give a triple- exponential upper bound and a double-exponential lower bound. We also do not have tight bounds for our more re-

stricted 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 [Abiteboul et al. 2000], and continuing through work of Spielmann [Spielmann 2003], Deutsch, Su and Vianu (e.g. [Deutsch et al. 2007]), Fritz et. al. [Fritz et al. 2009], and Deutsch et. al. [Deutsch et al. 2009]. 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 nondeterministic 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 [Abiteboul et al. 2000] 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 nontrivial ways. [Abiteboul et al. 2000] 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 [Abiteboul et al. 2000]). The technique applied is similar to that in Theorem 6.2, 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 [Spielmann 2003; Deutsch et al. 2007; Deutsch et al. 2009; Fritz et al. 2009] deal with transducers that can delete as well as insert into their internal state. A key restriction is *input-guardedness*, which insures decidability [Deutsch et al. 2007] – 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, [Deutsch et al. 2009] 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 [Damaggio et al. 2012]. [Hariri et al. 2013] obtains decidability for a different variant of the "artifact model", notably even for a branching-time logic. [Belardinelli et al. 2012] isolates conditions on artifact-based systems that imply the existence of finite-state abstractions (and hence decidability via a reduction to traditional model-checking).

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 [Ullman 1989] and Rajaraman et. al. [Rajaraman et al. 1995], continuing with Chang/Li's work in the early 2000s [Li and Chang 2001; Li 2003] Ludäscher/Nash's and Deutsch et. al.'s work in the mid-2000's [Nash and Ludäscher 2004; Deutsch et al. 2007] and Cali et. al. [Calì and Martinenghi 2008]. 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 [Calì and Martinenghi 2008], which establishes a coNEXPTIME upper bound for conjunctive queries. [Benedikt et al. 2011] proves a matching coNEXPTIME lower bound for containment for conjunctive queries, and a co-2NEXPTIME upper bound for positive queries. [Benedikt et al. 2011] 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 4.9. This is better than the prior bounds from [Calì and Martinenghi 2008; Benedikt et al. 2011].

Note that [Benedikt et al. 2011] also makes an erroneous claim: 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.

First order linear and modal logics. First order linear temporal logic (FOLTL) formulas are evaluated over linear sequences of first-order structures, mixing both temporal and first-order modal operators. A comprehensive reference for this can be found in Chapters 11 and 12 of [Kurucz et al. 2003]. As mentioned in the body of the paper, full FOLTL is easily seen to subsume all of the logics considered here, but undecidability is inherited of FOLTL from first-order logic. Prior work has shown much smaller sublogics that are undecidable. For example, in Theorem 11.3 of [Kurucz et al. 2003], the two-variable fragment with only monadic relations is shown undecidable. This does not subsume our undecidability result, given that the logic in Theorem 11.3 allows one to compare elements across arbitrary time points, while our logic only allows comparison at adjacent times, and only via the bindings of the access methods. In addition, the logic used to compare within a world in Theorem 11.3 allows arbitrary quantifier alternation, while our logic has only boolean combinations of positive existential formulas. Similar comments apply to undecidability for the Guarded Fragment of first-order temporal logic (Theorem 11.17 of [Kurucz et al. 2003]).

In terms of decidability results, prior work has also identified a family of decidable fragments, all lying within the Monodic fragment, in which Until formulas are allowed to have at most one free domain element variable: see [Hodkinson et al. 2000] or Chapter 11 of [Kurucz et al. 2003]. For example Theorem 11.12 of [Kurucz et al. 2003] shows that the two variable Monodic fragment is decidable. These results are easily seen not to imply our decidability results, for several reasons: first, our formalism allows an arbitrary arity in the free variables to be shared between time points

(in our case, consecutive time points), while in the two-variable Monodic fragment the number of shared variables is limited. Secondly, our results consider satisfiability not over arbitrary sequences, but those that successively reveal a single underlying model. Although this property of a sequence can be axiomatized in full FOLTL, it certainly cannot be axiomatized in the decidable fragments.

While the above results concern temporal logics, decidability and undecidability has also been studied for multi-dimensional modal logics, which are evaluated on structures connected by multiple binary relations, which are not a priori required to be linear orderings. For example, Chapters 5, 6, and 7 of [Kurucz et al. 2003] analyze multi-dimensional modal logics evaluated over structures built up as "products" of traditional Kripke structures, possibly satisfying additional axioms – e.g. one binary relation might be restricted to be a linear order.

Chapter 5 and 7 of [Kurucz et al. 2003] include a number of undecidability results concerning products. Some of the results exploit specific restrictions on the structures or the connecting binary relations. For example, Theorem 5.38 of [Kurucz et al. 2003] concerns undecidability where a product is taken using two linear orders, even when the individual worlds are propositional. These results do not subsume our undecidability result, because the logics allow arbitrary nesting of negation within a world and nesting of the multiple available modalities for relating two worlds. In contrast, our logic allows only a single un-nested reference to consecutive worlds, with only boolean combinations of positive existential formulas available at the world level.

There are also a number of decidability results for multi-dimensional modal logics based on products (see, e.g. Theorems 6.18, 6.24, 6.33 from Chapter 6 of [Kurucz et al. 2003]). All of these are easily seen not to subsume our main decidability result, since they deal only with underlying relational structures of arity 2, while our results deal with arbitrary arity. Naturally, some of the other reasons for incomparability cited above also hold here. For example, in the case of propositional worlds, decidability can be regained by allowing one of the connecting binary relations to be arbitrary (in contrast to Theorem 5.38 of [Kurucz et al. 2003] which restricts both relations to be linear orders). But naturally this does not subsume our decidability results, since our setting is not propositional, we have additional restrictions on the evolution of the structures (as mentioned above), and the connecting relations have additional shared parameters corresponding to the method binding.

Also in this line is the work of [Wolter and Zakharyaschev 1999], which gives decidability results for "modalized description logics", a variant of description logics appropriate for worlds that evolve via a binary relation. They likewise deal only with arity 2, so are incomparable with our decidability results. In addition, many of the decidability results (e.g. Theorem 22 of [Wolter and Zakharyaschev 1999]) do not apply to the chase where the binary relation (frame) is a linear order, which is the case of interest to us. Indeed [Wolter and Zakharyaschev 1999] also provides undecidability results for linear-ordered frames (e.g. Theorem 25). But again the logic is incomparable with the one used in our undecidability results, allowing nesting of modalities and negation.

Acknowledgements This work was funded by EPSRC EP/H017690/1, the Engineering and Physical Sciences Research Council UK. The second author was partially funded by the INRIA project Northern European associate teams between INRIA Lille and University of Oxford.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

ABITEBOUL, S., VIANU, V., FORDHAM, B. S., AND YESHA, Y. 2000. Relational transducers for electronic commerce. *Journal of Computer and System Sciences 61,* 2, 236–269.

BELARDINELLI, F., LOMUSCIO, A., AND PATRIZI, F. 2012. An abstraction technique for the verification of artifact-centric systems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference (KR)*.

BENEDIKT, M., GOTTLOB, G., AND SENELLART, P. 2011. Determining relevance of accesses at runtime. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 211–222.

CALÌ, A., CALVANESE, D., AND MARTINENGHI, D. 2009. Dynamic query optimization under access limitations and dependencies. *Journal of Universal Computer Science 15,* 1, 33–62.

CALÌ, A. AND MARTINENGHI, D. 2008. Conjunctive query containment under access limitations. In *International Conference on Conceptual Modeling (ER)*. 326–340.

CHANDRA, A. AND VARDI, M. Y. 1985. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing 14,* 3, 671–677.

CHAUDHURI, S. AND VARDI, M. Y. 1997a. On the equivalence of recursive and nonrecursive Datalog programs. *Journal of Computer and System Sciences 54,* 1, 61–78.

CHAUDHURI, S. AND VARDI, M. Y. 1997b. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences 54,* 1, 61–78.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 2000. *Model Checking*. MIT Press.

COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 1997. *Tree Automata Techniques and Applications*.

DAMAGGIO, E., DEUTSCH, A., AND VIANU, V. 2012. Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst. 37,* 3, 22:1–22:36.

DEUTSCH, A., HULL, R., PATRIZI, F., AND VIANU, V. 2009. Automatic verification of data-centric business processes. In *ICDT*. 252–267.

DEUTSCH, A., LUDÄSCHER, B., AND NASH, A. 2007. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Computer Science 371,* 3, 200–226.

DEUTSCH, A., SUI, L., AND VIANU, V. 2007. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences 73*, 442–474.

EMERSON, E. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Vol. B. MIT.

FRITZ, C., HULL, R., AND SU, J. 2009. Automatic construction of simple artifact-based business processes. In *ICDT*. 225–238.

HARIRI, B. B., CALVANESE, D., GIACOMO, G. D., DEUTSCH, A., AND MONTALI, M. 2013. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 163–174.

HODKINSON, I., WOLTER, F., AND ZAKHARYASCHEV, M. 2000. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic 106,* 1âĂŞ3, 85 – 134.

KURUCZ, A., WOLTER, F., ZAKHARYASCHEV, M., AND GABBAY, D. M. 2003. *Many-Dimensional Modal Logics: Theory and Applications*. Elsevier.

LI, C. 2003. Computing complete answers to queries in the presence of limited access patterns. *VLDB J. 12,* 3, 211–227.

LI, C. AND CHANG, E. Y. 2001. Answering queries with useful bindings. *ACM Transactions on Database Systems 26,* 3, 313–343.

NASH, A. AND LUDÄSCHER, B. 2004. Processing first-order queries under limited access patterns. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 307–318.

RAJARAMAN, A., SAGIV, Y., AND ULLMAN, J. D. 1995. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 105–112.

SPIELMANN, M. 2003. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences 66,* 1, 40–65.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems, V2*. Comp. Sci. Press.

WOLTER, F. AND ZAKHARYASCHEV, M. 1999. Modal description logics: Modalizing roles. *Fundamenta Informaticae 39,* 4, 411–438.