Datalog LITE: A deductive query language with linear time model checking

GEORG GOTTLOB
Vienna University of Technology
ERICH GRÄDEL
RWTH Aachen
and
HELMUT VEITH
Vienna University of Technology

We present Datalog LITE, a new deductive query language with a linear time model checking algorithm, i.e., linear time data complexity and program complexity. Datalog LITE is a variant of Datalog that uses stratified negation, restricted variable occurrences and a limited form of universal quantification in rule bodies.

Despite linear time evaluation, Datalog LITE is highly expressive: It encompasses popular modal and temporal logics such as CTL or the alternation-free μ-calculus. In fact these formalisms have natural presentations as fragments of Datalog LITE. Further Datalog LITE is equivalent to the alternation-free portion of guarded fixed point logic. Consequently, linear time model checking algorithms for all mentioned logics are obtained in a unified way.

The results are complemented by inexpressibility proofs to the effect that linear time fragments of *stratified* Datalog have too limited expressive power.

Categories and Subject Descriptors: H.2.3 [**Information Systems**]: Database Management—*Query Languages*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Modal Logic*; *Temporal Logic*; D.2.4 [**Software**]: Software/Program Verification—*Model Checking*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: databases, verification, temporal logics, guarded logics, complexity

---

Authors' addresses: Georg Gottlob and Helmut Veith, Institut für Informationssysteme, Abteilung für Datenbanken und Artificial Intelligence, Technische Universität Wien, A-1040 Wien, Austria. Email: {gottlob,veith}@dbai.tuwien.ac.at, http://www.dbai.tuwien.ac.at. Erich Grädel, Mathematische Grundlagen der Informatik, RWTH Aachen, D-52056 Aachen, Germany. Email: graedel@informatik.rwth-aachen.de, http://www-mgi.informatik.rwth-aachen.de.

This work was supported by the Austrian Science Fund Project N Z29-INF, by the Max Kade Foundation, and by Deutsche Forschungsgemeinschaft (DFG). Some of the results in the present paper have appeared without full proof in the shorter workshop paper [Gottlob et al. 2000].

1. INTRODUCTION

Databases and Datalog. Since the introduction of relational databases in the seventies [Codd 1972], the close relationship between database theory and formal logic, in particular finite model theory has been understood and exploited. In database theory, databases are often identified with finite relational structures. A *query* associates to each input database over a given schema (vocabulary) a result consisting of one or more output relations. Queries are formulated in *query languages*. *Datalog* (cf. [Abiteboul et al. 1995; Ceri et al. 1990; Ullman 1989] is a very powerful, well-studied declarative query language based on Horn clause logic. Datalog adapts the paradigm of Logic Programming to the database setting. Pure Datalog can be made more expressive by using *negated literals* in the rule bodies. For our purposes, there are two relevant extensions of pure Datalog by negation: Datalog with *stratified negation* [Apt et al. 1988] where negation does not interact with recursion, and the more expressive Datalog with *well-founded negation* [Gelder et al. 1991]. For an overview and introduction to database theory, please refer to [Abiteboul et al. 1995].

In general, query languages are more expressive and versatile than temporal logics such as CTL (see below). The trade-off is that even for relatively weak query languages like the relational calculus (alias SQL alias first-order logic alias recursion-free Datalog) the best known deterministic time bound is polynomial in the size of the structure (the so-called *data complexity*), and exponential in the length of the formula (*program complexity*). Linear time results are known for restricted classes of relational databases such as relational structures with bounded degree or bounded tree width [Courcelle 1990; Seese 1996]. In general, there is an exponential gap between data complexity and program complexity [Vardi 1982; Gottlob et al. 1999], and satisfiability is undecidable for most query languages.

Computer-Aided Verification. During the last decade, temporal logic model checking has become one of the preeminent applications of logic in computer science. Temporal logic model checking is a technique for verifying that a system satisfies its specification by (i) representing the system as a Kripke structure, (ii) writing the specification in a suitable temporal logic, and (iii) algorithmically checking that the Kripke structure is a model of the specification formula. Model checking has been successfully applied in hardware verification, and is emerging as an industrial standard tool for hardware design. For recent overviews of model checking, the reader is referred to [Clarke et al. 2000; Clarke and Schlingloff 2000].

Early research on the temporal logics used in model checking has concentrated on properties that are crucial for the method, distinguish them from many other logics in computer science, and serve as a basis for more sophisticated techniques such as abstraction. These properties are decidability of the satisfiability problem, fast – often linear time – model checking procedures, and structure theorems about the models of temporal formulas such as the tree model property and the finite model property. Some of the most prominent examples of temporal logics are branching time logics such as computation tree logic CTL [Clarke and Emerson 1981], the modal μ-calculus L_μ [Kozen 1983], and its alternation-free fragment L_{μ_1}.

The main practical problem in model checking is the so-called state explosion problem caused by the fact that the Kripke structure represents the *state space* of the system under investigation, and thus it is of size exponential in the size of the system description. The state explosion problem is alleviated by a number of techniques, in particular symbolic techniques which represent the Kripke structure by binary decision diagrams [Bryant 1986; Burch et al. 1992; McMillan 1993] or Boolean Functions [Biere et al. 1999], abstraction techniques (e.g. [Clarke et al. 1994; Kurshan 1994; Clarke et al. 2000]), and reduction techniques [Peled et al. 1997]. Note that symbolic techniques rely on effective translations of temporal logic into μ-calculus, i.e., fixed point logic.

The model checking problem in databases and computer-aided verification. Databases and computer-aided verification share a common interest in the problem of evaluating logical formulas (specifications, queries) over finite structures (Kripke structures, relational databases). This problem is commonly called the model checking problem. In both fields, data complexity is considered to be more relevant than program complexity, i.e., the practical complexity of model checking is related to the size of the typically large structure rather than to the size of the formulas. Despite these abstract similarities, the practical requirements of the fields lead to a different focus on the model checking problem.

First of all, due to the state explosion problem, the size of the structures in verification is by several orders of magnitude greater than even for large databases. Researchers and engineers may spend months trying to devise techniques which make the model checking problem for a *single structure* feasible. If the verification algorithm terminates in any reasonable amount of time, the verification is considered successful.

On the other hand, database applications require scalability and versatile query languages that have predictable and fast query answer times. The finite structure (database) is typically smaller than in verification, but subject to continuous updates which make many verification-style preprocessing techniques like abstraction and symbolic representation infeasible.

Temporal logics possess many of the properties needed for query languages, but are semantically too restricted to be used as query languages. This motivates the design and investigation of hybrid query languages which are sufficiently versatile to be used in a database context, but retain the favorable properties of temporal logic.

Contribution of this Paper. In this paper we investigate query languages that (1) encompass a natural recursion mechanism, (2) extend the expressive power of temporal logics such as CTL and the alternation-free μ-calculus, (3) have linear time data and program complexity, and (4) are applicable to arbitrary relational structures. The result of our analysis is the language Datalog LITE – **LI**near **T**ime Datalog with **E**xtensions. Datalog LITE is a deductive query language that simultaneously inherits the intuitive semantics of Datalog and the favorable properties of temporal logics, in particular linear time model checking. Datalog LITE is intended as a first step in comparing and combining methods from two quite distinct fields. This paper is mainly concerned with fundamental logical and algorithmic properties of Datalog LITE.

2. RESEARCH PROGRAMME, RESULTS AND RELATED WORK

Our goal was to identify a deductive query language – a suitable variant of Datalog – with the following properties:

— The language should be clearly syntactically identifiable, and ample enough to allow the programmer to make use of the main paradigms of Logic Programming (including in particular some form of negation).

— The language should have linear time data complexity. This means that the time complexity of evaluating any fixed query over variable input structures is $O(n)$, where n is the size of a suitable encoding of the input structure.

— The language should also have low (preferably linear time) program complexity, i.e. one should be able to evaluate programs in linear time with respect to the length of the program.

— It should be versatile enough to formulate queries over arbitrary finite relational schemas.

— It should be expressive enough to state a large number of useful temporal properties. In particular, we require that our new Datalog variant be at least as expressive as CTL.

— Given the increasing importance of model checking on infinite transition systems, the language should be able to express relevant temporal properties not just on finite Kripke structures but also on infinite ones.

Note that an *ad hoc* translation of CTL into Stratified Datalog (as outlined in Section 3.2) does *not* satisfy these requirements. The fragment suggested by such a translation is syntactically rather limited, since it only contains unary and binary relations. Further, as we will prove, such a translation can be valid only over finite structures, but not over infinite ones. Finally the resulting fragment is not known to admit a linear time evaluation algorithm.

In this paper, we present *Datalog LITE*, a variant of Datalog that fulfills all desired criteria.

Linear Time Datalog. We first define the fragment *Datalog LIT* (LIT stands for linear time) as the set of all stratified Datalog queries (over arbitrary input structures) whose rules are either monadic or guarded. A *monadic* rule is a rule that contains only monadic literals. A rule is *guarded*[1] if its body contains a positive atom (the *guard*) in which all variables of the rule occur.

EXAMPLE 2.1. *The two types of Datalog LIT rules (i.e., monadic and guarded rules), are exemplified by the following rules:*
$Tx \leftarrow Eyy, \neg Fx$
$Tx \leftarrow Exy, Ty$
On the other hand, the rule $Txy \leftarrow Exz, Tzy$ *is not guarded.*

We prove that Datalog LIT has linear time data complexity (Theorem 4.4). Moreover, for bounded arity programs or for programs whose guards are all input relations, also the combined complexity is in linear time.

Although Datalog LIT can express a large number of interesting temporal properties, the language is not powerful enough for expressing full CTL. For example, (as shown in Section 6, Theorem 6.4), the simple CTL formula $\mathbf{AF}\varphi$ ("on all paths, eventually φ") cannot be expressed in Datalog LIT. Consequently, we define an extended language *Datalog LITE*. The latter is obtained from Datalog LIT by including the possibility of using a limited (guarded) kind of universal quantification in the rule bodies.

A *generalized literal* G is an expression of the form $(\forall y_1 \cdots y_n.\alpha)\beta$ where α and β are atoms, and the free variables in β also occur in α. The intended meaning of $(\forall y_1 \cdots y_n.\alpha)\beta$

[1] This notion is completely unrelated to the concept of *guarded Horn clause* as defined in [Murakami 1990].

is its standard first order semantics, that is, $\forall y_1 \cdots y_n(\alpha \to \beta)$. (For a precise definition, see Section 4.1.)

EXAMPLE 2.2. *In addition to Datalog LIT rules, Datalog LITE has rules with generalized literals like in the following one-rule program* Π*:*

$$Wx \leftarrow (\forall z Exz)Wz$$

Suppose that Exy *means "There is a link from URL* x *to URL* y*". Then* (Π, W) *computes the well-founded elements of relation* E*, i.e., those "good" URLs from which a recursive web-suck will terminate. Note that* Π *encodes the natural algorithm: in the first step, all* x *with no outgoing links are collected in* W*. (For all such* x*,* $(\forall z Exz)\varphi$ *is vacuously true independent of* φ*.) In the following steps, we add a new URL to the set* W *of known good URLs, if all reachable URLs are known to be good.*

Results. We prove the following relevant facts about Datalog LITE: (An overview of the results is given in Figure 1.)

(i) Every *fixed* Datalog LITE program admits linear time model checking. (Theorem 4.10).
This remains true even for variable Datalog LITE programs in the cases where the arities of the input predicates are bounded or only input atoms can be used as guards. We develop and describe a linear-time model checking algorithm in Section 4.1. The two main ingredients for the proofs are *locality (Gaifman) invariance* of Datalog LITE, and a reduction to the satisfiability problem for propositional Horn formulae (or equivalently, propositional unit resolution).
Thus, our algorithm is based on a linear time solvable fragment of the propositional satifsfiability problem; note that SAT procedures have also found applications in model checking [Biere et al. 1999].

(ii) We show that, semantically, every Datalog LITE program is equivalent to a Datalog LITE program where all guards are input atoms; for arbitrary programs, the translation requires exponential time. In fact, evaluating variable Datalog LITE programs with unbounded predicate arities is EXPTIME- complete. (Proposition 4.17).

(iii) Datalog LITE is equivalent in expressive power to alternation-free guarded fixed point logic, a natural fragment of the guarded fixed point logic μGF which has recently been studied by Grädel and Walukiewicz [Grädel and Walukiewicz 1999]; several important properties like satisfiability in EXPTIME and a generalized tree model property are thus obtained for Datalog LITE.

(iv) Well-known logical formalisms such as propositional multi-modal logic (ML), computation tree logic (CTL), the alternation free modal μ-calculus (L_{μ_1}), and guarded first order logic (GF) each correspond to well-defined and syntactically simple fragments of Datalog LITE. (These fragments will be defined in Section 7.) Consequently, we obtain a unified proof for linear time model checking for many formalisms.

(v) Datalog LITE (with bounded arities) is strictly more expressive than CTL and the alternation-free μ-calculus L_{μ_1}. For example, it is obvious that *past modalities* (cf. [Vardi 1998]) are expressible in Datalog LITE. Furthermore, Datalog LITE can

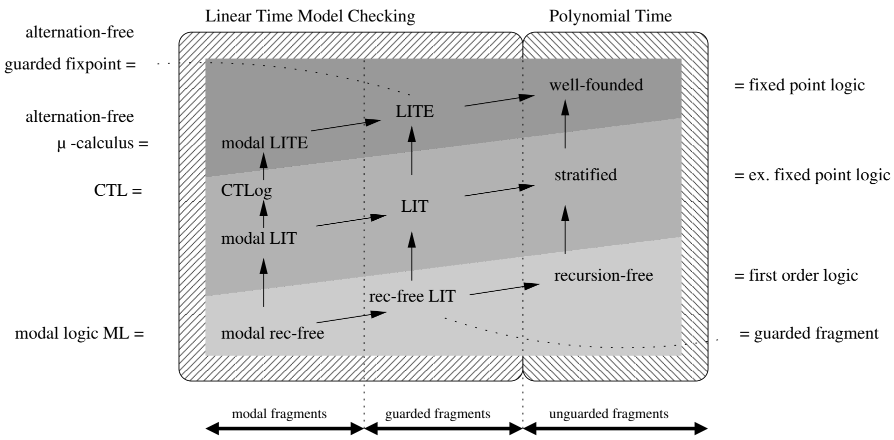

Fig. 1. Overview of Datalog fragments studied in this paper. In the figure, all logics inside the box are versions of Datalog. LIT and LITE stand for Datalog LIT and Datalog LITE respectively.

define new modalities, e.g. via Boolean combinations of existing ones, or describe regular patterns along paths, similar as ETL [Wolper 1983; Vardi and Wolper 1994] does in a non-branching framework.

Remark 2.3 – *Datalog LITE and well-founded negation.* Note that generalized literals are not just artificial "plug ins" to Datalog. They can be expressed in Datalog with well-founded negation. In fact, universal quantification can be easily resolved by double negation under the well-founded semantics. Thus Datalog LITE can be considered a fragment of Datalog with well-founded negation. Note however, that the semantics of Datalog LITE is a natural stratified semantics and does not use complicated and less intuitive constructions as in the case of well-founded or stable semantics of Datalog with negation.

2.1 Related work.

Branching Time Temporal Logic and Logic Programming. It is not new that CTL, the modal μ-calculus and related formalisms can be translated into logic programming. Such translations were carried-out in [Charatonik and Podelski 1998; Ramakrishnan et al. 1997; Cui et al. 1998]. In a recent paper by Immerman and Vardi [Immerman and Vardi 1997] it is shown how various temporal logics can be translated into transitive closure logic (FO+TC). Since transitive closure logic is equivalent to a natural fragment of Stratified Datalog, this approach yields a translation of temporal logics into Stratified Datalog. The translation only works on finite structures.

Note, however, that all previously proposed translations are ad hoc translations of existing modal model-checking formalisms. None of them identifies an ample fragment of Datalog with guaranteed linear time data complexity suited for model checking. To our best knowledge, the first such fragment is Datalog LITE as introduced in the present paper.

Datalog LITE and guarded fixed point logic. The definitions of guarded rules and generalized literals are motivated by the notion of *guarded quantification* as introduced by Andrèka, van Benthem and Nèmeti [Andréka et al. 1998] (see Section 8). The variables that appear in the body but not in the head of a Datalog rule are implicitly existentially quantified. The condition that all variables of the rule appear in one atom implies that this existential quantification is guarded in the sense of [Andréka et al. 1998]. Similarly the variable occurrence condition in the definition of generalized literals implies that the universal quantifier is guarded in this sense. In fact we will see that Datalog LITE can be viewed as a clausal presentation of the alternation-free fragment of guarded fixed point logic [Grädel and Walukiewicz 1999].

Datalog LITE and LTL. When viewed as a temporal logic, Datalog LITE clearly is a branching time logic. In linear time logics such as LTL, time is typically modeled by the natural numbers or the integers, i.e., an infinite linear Kripke structure. (Note that here *linear* refers to the non-branching structure of the time-line, and not to time complexity.) Given a Kripke structure, linear Kripke structures can be obtained by unwinding the Kripke structure into an infinite tree, and considering all infinite paths separately. An LTL formula is defined to be true over a Kripke structure, if it is true on *all* infinite paths. Under this translation, LTL and CTL are incomparable, and we expect that the same holds true for LTL and Datalog LITE. We leave a rigorous investigation of Datalog LITE in a linear time framework to future work, cf. Section 9.

The situation is different when we consider the expressive power of logics over single linear (non-branching) Kripke structures. It is easy to see that Datalog LIT rules can simulate finite automata over linear Kripke structures. Since Datalog LIT is closed under complement and nesting of subprograms, the results of Vardi and Wolper [Vardi and Wolper 1994, Section 6] imply that over (possibly infinite) linear Kripke structures, Datalog LIT has the same expressive power as ETL, and subsumes LTL. Note that over linear Kripke structures, the guarded quantifier of Datalog LITE is redundant.

Deductive Temporal Databases. Linear time logics have also been studied in a database context. In particular, Chomicki and Imieliński [Chomicki and Imieliński 1988; 1989; Chomicki 1990b; 1990a] as well as Abadi, Manna and Baudinet [Abadi and Manna 1989; Baudinet 1989b; 1989a; 1995; 1992] have studied Datalog$_{1S}$ and Templog, two semantically equivalent variants of Datalog [Baudinet et al. 1993]. (In the following, we will refer to this logic as Datalog$_{1S}$.)

Both use a framework, where linear time is encoded by the natural numbers (i.e., timestamps), and each database relation is two-sorted with time as an additional argument. For a fixed time point, Datalog$_{1S}$ behaves like ordinary Datalog. (Both negation-free and stratified variants of Datalog$_{1S}$ have been investigated.) When each time instance of the database is viewed as a symbol in a large finite alphabet Σ, then Datalog$_{1S}$ can be viewed as a linear time logic over a linear Kripke structure, where each state is labelled by an element of Σ. From this point of view, Datalog$_{1S}$ is subsumed by ETL, cf. [Baudinet et al. 1993; Baudinet 1989a; 1989b], and also by Datalog LIT. For a more thorough overview about temporal deductive databases, please refer to [Baudinet et al. 1993].

3. BACKGROUND ON TEMPORAL LOGIC AND DATALOG

3.1 Modal and temporal logics

In this subsection, we recall the definitions of some logics commonly used in verification. For more background on temporal logics, the reader is referred to [Emerson 1990].

Kripke structures. The logics we describe here are evaluated on Kripke structures (with one or more transition relations) at a particular state. A Kripke structure (or transition system) for a finite set A of 'actions' is a relational structure $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, P_2, \dots, P_k)$ whose universe S is the set of *states*, with unary relations P_i describing properties of states (atomic propositions) and binary relations $E_a \subseteq S \times S$ describing transitions between states (actions). Given a formula ψ and a Kripke structure $\mathcal{K}$ with state v, we will write $\mathcal{K}, v \models \psi$ to denote that the formula ψ holds in the Kripke structure $\mathcal{K}$ at state v. Therefore such formulae are sometimes called state formulae. A state formula ψ can also be understood as a (monadic) query that associates with each Kripke structure $\mathcal{K}$ the set of states v such that $\mathcal{K}, v \models \psi$.

Propositional modal logic. The simplest of these formalisms is propositional modal logic ML. We describe it for the case of several transition relations, i.e. for reasoning about Kripke structures $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, P_2, \dots, P_k)$.

Syntax of ML . The formulae of ML are defined by the following rules.

(S1) Each atomic proposition P_i is a formula.

(S2) If ψ and φ are formulae of ML, then so are $(\psi \wedge \varphi)$ and $\neg\psi$.

(S3) If ψ is a formula of ML and $a \in A$ is an action, then $\langle a\rangle\psi$ and $[a]\psi$ are formulae of ML.

If there is only one action (i.e., only one transition relation E in the Kripke structure) one simply writes $\Box$ and $\Diamond$ for the modal operators.

Semantics of ML . Let ψ be a formula of ML, $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, P_2, \dots, P_k)$ a Kripke structure, and $v \in S$ a state. In the case of atomic propositions, $\psi = P_i$, we have $\mathcal{K}, v \models \varphi$ iff $v \in P_i$. Boolean connectives are treated in the natural way. Finally for the semantics of the modal operators we put

$$\mathcal{K}, v \models \langle a\rangle\psi \text{ iff } \mathcal{K}, w \models \psi \text{ for } \textit{some } w \text{ such that } (v, w) \in E_a$$
$$\mathcal{K}, v \models [a]\psi \text{ iff } \mathcal{K}, w \models \psi \text{ for } \textit{all } w \text{ such that } (v, w) \in E_a$$

CTL and CTL*. The logics CTL (computation tree logic) and CTL* extend ML by path quantification and temporal operators on paths. We define first CTL*, and obtain CTL as a fragment of CTL*. For computation tree logics, one usually restricts attention to Kripke structures with a single binary relation which is required to be *total*, i.e. for every state v there exists a least one state w that is reachable from v.

Syntax of CTL*. We have state formulae and path formulae in CTL*. The path formulae are auxiliary formulae to make the inductive definition more transparent. Actually we are interested in state formulae. The state formulae are defined by the rules (S1) and (S2) (read everywhere state formula instead of formula) and the rule

(S4) If ϑ is a path formula, then $\mathbf{E}\vartheta$ and $\mathbf{A}\vartheta$ are state formulae.

The path formulae are defined by

(P1) Each state formula is a path formula.

(P2) If ϑ, η are path formulae, then so are $(\vartheta \wedge \eta)$ and $\neg\vartheta$.

(P3) If ϑ, η are path formulae, then so are $\mathbf{X}\vartheta$ and $(\vartheta\mathbf{U}\eta)$.

Semantics of CTL*. Consider a total Kripke structure $\mathcal{K} = (S, E, P_1, P_2, \dots, P_k)$. An *infinite E-path* (or just an infinite path) is an infinite sequence $\bar{s} = s_0, s_1, s_2, \dots$ of states such that for all $i \geq 0$, $(s_i, s_{i+1}) \in E$ for all i. $\bar{s}^i$ denotes the suffix path $s_i, s_{i+1}, s_{i+2}, \dots$. As above, we write $\mathcal{K}, s_0 \models \psi$ to denote that the state formula ψ is true in $\mathcal{K}$ at state s_0. We write $\mathcal{K}, \bar{s} \models \vartheta$ to denote that the path formula ϑ is true on the infinite path $\bar{s}$ in $\mathcal{K}$. We define $\models$ inductively as follows. For formulae defined via rules (S1), (S2) and (P2) the meaning is obvious.

(S4) $\mathcal{K}, s_0 \models \mathbf{E}\vartheta$ iff there exists an infinite path $\bar{s}$ from s_0 such that $\mathcal{K}, \bar{s} \models \vartheta$.
$\mathcal{K}, s_0 \models \mathbf{A}\vartheta$ iff $\mathcal{K}, \bar{s} \models \vartheta$ for all infinite paths $\bar{s}$ that start at s_0.

(P1) $\mathcal{K}, \bar{s} \models \psi$ iff $\mathcal{K}, s_0 \models \psi$ where s_0 is the starting point of $\bar{s}$.

(P3) $\mathcal{K}, \bar{s} \models (\vartheta \mathbf{U} \eta)$ iff for some $i \geq 0$, $\mathcal{K}, \bar{s}^i \models \eta$ and $\mathcal{K}, \bar{s}^j \models \vartheta$ for all $j < i$.
$\mathcal{K}, \bar{s} \models \mathbf{X}\vartheta$ iff $\mathcal{K}, \bar{s}^1 \models \vartheta$.

CTL is the fragment of CTL* without nestings and without Boolean combinations of path formulae. Thus, rules (P1) – (P3) are replaced by

(P0) If ψ, φ are state formulae, then $\mathbf{X}\psi$ and $(\psi \mathbf{U} \varphi)$ are path formulae.

Actually, for CTL we can disregard path formulae altogether and equivalently define the set of state formulae by the rules (S1), (S2) of ML together with rules

(S3') If ψ is a (state) formula, then so are $\mathbf{EX}\psi$ and $\mathbf{AX}\psi$.

(S4) If ψ and φ are (state) formula, then so are $\mathbf{E}(\psi \mathbf{U} \varphi)$ and $\mathbf{A}(\psi \mathbf{U} \varphi)$.

Note that indeed the formulae $\mathbf{EX}\psi$ and $\mathbf{AX}\psi$ defined by (S3') are equivalent to $\Diamond\psi$ and $\Box\psi$, respectively.

Other temporal operators are defined by abbreviations as follows: $\mathbf{F}\psi$ ("eventually ψ") abbreviates $(\mathrm{true}\mathbf{U}\psi)$ and $\mathbf{G}\psi$ ("globally ψ") abbreviates $\neg\mathbf{F}\neg\psi$. Hence one can form in CTL formulae of the form $\mathbf{EF}\psi$, $\mathbf{AF}\psi$, $\mathbf{EG}\psi$ and $\mathbf{AG}\psi$. Well-foundedness statements such as $\mathbf{AF}\psi$ ("on all paths, eventually ψ") will be of special importance to us.

The μ-calculus L_μ. The propositional μ-calculus L_μ is propositional modal logic augmented with least and greatest fixed points. It subsumes almost all of the commonly used modal logics, in particular LTL, CTL, CTL*, PDL and also many logics used in other areas of computer science, for instance most description logics.

Syntax of L_μ. The μ-calculus extends propositional modal logic ML (including propositional variables $X, Y, \dots$) by the following rule for building fixed point formulae.

(Sμ) If ψ is a formula in L_μ, and X is a propositional variable that does not occur negatively in ψ, then $\mu X.\psi$ and $\nu X.\psi$ are L_μ formulae.

The *alternation-free fragment* of L_μ does not permit 'genuine' nestings of least and greatest fixed points. This means that that in formulae of the form $\mu X.\psi$ (respectively $\nu X.\psi$), there is no subformula $\nu Y.\varphi$ (respectively $\mu Y.\varphi$) such that the outer fixed point variable X occurs inside φ. Note, however, that nestings of the form $\mu X.(\varphi(X) \vee \nu Y.\vartheta(Y))$ are allowed provided that X does not appear in ϑ.

Semantics of L_μ. The semantics of the μ calculus is given as follows. A formula $\psi(X)$ with propositional variable X defines on every Kripke structure $\mathcal{K}$ (with state set S) an operator $\psi^{\mathcal{K}} : 2^S \to 2^S$ assigning to every set $X \subseteq S$ the set

$$\psi^{\mathcal{K}}(X) := \{s \in S : (\mathcal{K}, X), s \models \psi\}.$$

If X occurs only positively in ψ, then $\psi^{\mathcal{K}}$ is *monotone* for every $\mathcal{K}$, and therefore has a least and a greatest fixed point. Now we put

(Sμ) $\mathcal{K}, s \models \mu X.\psi$ iff s is contained in the least fixed point of the operator $\psi^{\mathcal{K}}$. Similarly $\mathcal{K}, s \models \nu X.\psi$ iff s is contained in the greatest fixed point of $\psi^{\mathcal{K}}$.

Remark 3.1. There is the following duality between least and greatest fixed points.

$$\nu X.\psi \equiv \neg\mu X.\neg\psi[X/\neg X].$$

Hence we can always eliminate greatest fixed points at the expense of introducing negations. If we do this with a formula in the alternation-free μ-calculus then in subformulae of form $\mu X.\varphi$, the variable X will not appear in the scope of a negation sign.

Rather than eliminating greatest fixed points it is often more convenient to keep them and instead push all negations to the atomic propositions (formulae in negation normal form).

Remark 3.2. It is not difficult to see that CTL formulae can equivalently be expressed by alternation-free formulae in the μ-calculus. Indeed, formulae $\mathbf{E}(\psi\mathbf{U}\varphi)$ and $\mathbf{A}(\psi\mathbf{U}\varphi)$ defined by rule (S4) of CTL are equivalent, respectively, to $\mu X.\varphi \vee (\psi \wedge \Diamond X)$ and $\mu X.\varphi \vee (\psi \wedge \Box X)$.

3.2 Datalog and stratified Datalog

Definition 3.3. A *Datalog rule* is an expression of the form $H \leftarrow B_1, \dots, B_r$ where H, the *head* of the rule, is an atomic formula $Ru_1 \cdots u_s$, and $B_1, \dots, B_r$, the *body* of the rule, is a collection of literals (i.e. atoms or negated atoms) of the form $Sv_1 \cdots v_t$ or $\neg Sv_1 \cdots v_t$ where $u_1, \dots, u_s, v_1, \dots, v_t$ are variables. The relation symbol R is called the *head predicate* of the rule.

A *basic Datalog-program* Π is a finite collection of rules such that none of its head predicates occurs negated in the body of any rule. The predicates that appear only in the bodies of the rules are called *extensional* or *input predicates*. Given a relational structure $\mathfrak{A}$ over the vocabulary of the input predicates, the program computes, via the usual fixed point semantics, an interpretation for the head predicates. A Datalog query is a pair (Π, Q) consisting of a Datalog program Π and a designated head predicate Q of Π. With every structure $\mathfrak{A}$, the query (Π, Q) associates the result $(\Pi, Q)[\mathfrak{A}]$, i.e. the interpretation of Q as computed by Π on input $\mathfrak{A}$. For details, please refer to [Abiteboul et al. 1995].

Expressing temporal properties in Datalog. It is easy to see that a Kripke structure $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, \dots, P_k)$ can be viewed as a relational structure, and consequently, Datalog queries can be issued over Kripke structures. The following example shows that interesting temporal properties can be expressed this way.

EXAMPLE 3.4. *Consider a query (Π, H) over relational Kripke structures (S, E, P, Q), where Π consists of the rules*

$$Hx \leftarrow Qx \qquad\qquad Hx \leftarrow \neg Px, Exy, Hy.$$

For any state $s \in S$, this query outputs Hs iff either Q holds on s or a state where Q holds is reachable from s by a path where $\neg P$ holds until Q is satisfied. Thus, on total relations E, the output relation H contains all states satisfying the CTL formula $\mathbf{E}(\neg P \mathbf{U} Q)$.

A Datalog rule is *linear* if at most one of the literals in the body contains a head predicate. A basic Datalog program is linear if all its rules are linear. For instance the program exhibited above for the formula $\mathbf{E}(\neg P \mathbf{U} Q)$ is clearly linear. (Note that the concept of linearity is essentially the same as *tail recursion* in PROLOG.)

Definition 3.5. A *stratified Datalog program* is a sequence $\Pi = (\Pi_0, \dots, \Pi_r)$ of basic Datalog programs which are called the *strata* of Π, such that each of the head predicates of Π is a head predicate in precisely one stratum Π_i and is used as an extensional predicate only in higher strata Π_j for $j > i$. In particular, this means that

(i) if a head predicate of stratum Π_j occurs *positively* in the body of a rule of stratum Π_i, then $j \leq i$, and

(ii) if a head predicate of stratum Π_j occurs *negatively* in the body a rule of stratum Π_i, then $j < i$.

The semantics of a stratified program is defined stratum per stratum. The extensional predicates of a stratum Π_i are either extensional in the entire program Π or are head predicates of a lower stratum. Hence, once the lower strata are evaluated, we can compute the interpretation of the head predicates of Π_i as in the case of basic Datalog-programs.

A stratified Datalog program is *linear* if in the body of each rule there is at most one occurrence of a head predicate of the same stratum (but there may be arbitrary many occurrences of head predicates from lower strata).

EXAMPLE 3.6. *Consider the CTL-formula $\psi := Q \wedge \mathbf{AF}P$ which is equivalent to the L_μ-formula $Q \wedge \mu X.P \vee \Box X$. We present a stratified linear Datalog program Π with two strata and designated head predicate H_ψ such that the result of the query (Π, H_ψ) on every* finite *total Kripke structure $\mathcal{K} = (V, E, P, Q)$ is the set of states $v \in V$ such that $\mathcal{K}, v \models \psi$. Π contains the rules*

$$
\begin{array}{ll}
Txy \leftarrow \neg Px, Exy & \qquad Txz \leftarrow Txy, \neg Py, Eyz \\
Sx \leftarrow Txx & \qquad Sx \leftarrow \neg Px, Exy, Sy \\
H_\psi x \leftarrow Qx, \neg Sx &
\end{array}
$$

The first stratum computes the set T of all pairs of states (u, v) such that there exists a path $u = u_0, u_1, \dots, u_m = v$ from u to v such that $K, u_i \models \neg P$ for all $i < m$, and the set S of all states from which there exists an infinite path on which $\neg P$ globally holds. Hence $v \in S$ iff $\mathcal{K} \models \neg\mathbf{AF}P$. Here the finiteness of the Kripke structure is used in an essential way, because only this guarantees that every infinite path eventually reaches a cycle. The second stratum computes H_ψ by taking the conjunction of Q with the complement of S.

PROPOSITION 3.7. *Over finite structures, each CTL-formula is equivalent to a stratified linear Datalog program.*

Using the examples presented above, this can be easily proved by induction. It also follows from the facts that, over finite structures, CTL can be embedded into transitive closure logic FO(TC) [Immerman and Vardi 1997] and that transitive closure logic has the same expressive power as stratified linear Datalog programs [Consens and Mendelzon 1993; Grädel 1992].

However, there are two drawbacks of this translation of CTL into stratified Datalog. In section 6, we will show that there is a conceptual mismatch between stratified Datalog and CTL:

(1) The translation is valid for finite Kripke structures only. Even over countable structures, it is impossible to express well-foundedness statements of the form $\mathbf{AF}\varphi$ by a stratified Datalog program (see Theorem 6.3 below).

(2) Even over finite structures, the translation is unsatisfactory since it uses binary head predicates in an essential way, cf. Theorems 6.4 and 5.3. We will show that stratified programs with only monadic head predicates are not sufficient for capturing all of CTL. We hence end up in a portion of Datalog for which it is not known whether it admits a linear time evaluation algorithm. On the other side it is known that CTL admits model checking algorithms that are linear both with respect to the size of the CTL-formula and the size of the Kripke structure.

4. LINEAR TIME MODEL CHECKING FOR DATALOG LIT AND DATALOG LITE

4.1 Datalog LIT

Definition 4.1. A Datalog rule is *monadic* if each of its literals has at most one free variable. Note that this does not necessarily imply that only unary predicates are used. We permit the appearance of literals $(\neg)Sv_1 \cdots v_k$ with $k > 1$ that may contain repeated occurrences of a single variable. A Datalog rule is *guarded* if its body contains a positive atom (the *guard* of the rule) in which all free variables of the rule occur. A Datalog LIT program is a stratified Datalog program whose rules are either monadic or guarded.

EXAMPLE 4.2. *The following Datalog LIT query (Π, T) computes the transitive closure of the binary relation E, restricted to paths controlled by a ternary relation C.*

$$\begin{aligned} Txy &\leftarrow Exy \\ Txz &\leftarrow Cxyz, Exy, Tyz. \end{aligned}$$

We immediately see that if C is the complete ternary relation, then T in fact is the transitive closure of E.

4.2 Linear time model checking for Datalog LIT

The most important property of Datalog LIT is that with respect to the time complexity of model checking, it is as easy as CTL, i.e. linear in the size of the specification and the input. Before stating this result, we have to make precise the notions of length and input representation.

As usual in verification, we adopt the convention that the input relations are given by lists of tuples. We shall refer to this representation as the *list representation* of the input structure.

Definition 4.3. Let $\mathfrak{A}$ be a relational structure. Then the size $|\mathfrak{A}|$ of $\mathfrak{A}$ is the sum of the sizes of the tuples contained in the relations of $\mathfrak{A}$, plus the number of domain elements.

In our algorithm we shall also use the *array representation*, which is more common in finite model theory. There, the characteristic membership function of a relation is stored in an array whose dimension equals the arity of the relation. (For graphs, this means that the graph is described by its adjacency matrix.) With array representation, membership of an individual tuple can be checked in constant time on Random Access Machines (cf. [van Emde Boas 1990b]), but on the other hand, we obtain a non-realistic notion of linear time computability when the adjacency matrix is sparse. However, it is easy to see that a Random Access Machine can translate the list representation into array representation in linear time. (Note that the standard RAM model [van Emde Boas 1990a] assumes an array to be initialised by zeroes. Alternatively, the initialization can be simulated by the *lazy array* technique as described in [Moret and Shapiro 1990] with time overhead linear in $|\mathfrak{A}|$.)

A propositional Datalog program is a program where no rule contains free variables. Thus, every rule is equivalent to a propositional Horn clause of the form $h \vee \neg b_1 \vee \cdots \vee \neg b_n$. It is well-known [Itai and Makowsky 1987; Dowling and Gallier 1984; Minoux 1988] that propositional Horn clause programs can be evaluated in *linear time* on RAMs. We shall refer to this linear time algorithm as **EvalHORN**. **EvalHORN** makes use of a special data structure to store the Horn clauses and the result in such a way that literals can be stored, deleted and read in constant time. We shall call such a data structure a *Horn clause base* (HCB), and use it in our algorithm.

A standard way to evaluate a negation-free Datalog program Π over an input structure $\mathfrak{A}$ is called *grounding*. This means that the language is extended by constant symbols for all domain elements from $\mathfrak{A}$, and the program is replaced by the rules obtained by instantiating constant symbols for the variables. The Gelfond-Lifschitz transform $GL(\Pi, \mathfrak{A})$ over $\mathfrak{A}$ then is the set of rules obtained from the set of ground instantiations of rules in Π, such that input literals which are true over $\mathfrak{A}$ are removed, and rules containing false input literals are removed completely.

The resulting propositional program is of polynomial size and can therefore be evaluated in polynomial time by **EvalHORN**. This procedure can be extended to stratified programs by evaluating the strata one by one [Berman et al. 1995].

THEOREM 4.4. *The time complexity of model checking (i.e., evaluation) of Datalog LIT is*

- *(a) linear in the input size and the program size for programs where all guards are input relations.*
- *(b) linear in the input size and the program size for bounded arity programs.*
- *(c) linear in the input size and exponential in the program size for arbitrary programs.*

PROOF. (a) The algorithm proceeds in two phases:
(1) Monadic rules are transformed in such a way that every rule contains at most one variable. This can be easily achieved by introducing new rules with variable-free heads. If there is a e.g. a rule $Fx \leftarrow Gx, Hy$, it will be replaced by $Fx \leftarrow Gx, H_y$, and $H_y \leftarrow Hy$. This translation increases the size of the program only by a constant factor, and can be done in linear time by a subprogram **Normalize**. **Normalize** guarantees that the number of ground instances of monadic rules is bounded by $|\mathfrak{A}|$.

(2) The Datalog program is evaluated stratum by stratum as a propositional Horn program similarly as sketched in [Berman et al. 1995]. Since the ground instances of a guarded rule are determined by the ground substitutions of the variables in the guard, the number of ground instantiations of a guarded rule is bounded by the number of tuples in the guard relation, and thus by $|\mathfrak{A}|$.

We conclude that for every rule, the number of ground instances is bounded by $|\mathfrak{A}|$, and obtain the algorithm **EvalLIT** which performs the grounding stratum by stratum.

Algorithm EvalLIT$(\Pi, \mathfrak{A})$

Normalize(Π)
store $\mathfrak{A}$ **in array** A
for all strata Π_i **from** Π_0 **to** Π_n
    for all rules $r \in \Pi_i$
        let $G(\bar{y}) :=$ **guard of** r
        for all tuples $\bar{d} \in G$
            $\sigma :=$ **unifier of** $\bar{d}$ **and** $\bar{y}$
            $r' := GL(\{r\sigma\}, A)$
            store r' **in HCB** H_i
    call EvalHORN(H_i)
    and store results in array A

It is easy to see from the above arguments that the algorithm is linear both in the program size and $|\mathfrak{A}|$.

The proof of cases (b) and (c) is deferred to Corollary 4.15 where is is shown that guards which are not input relations can be eliminated in time exponential in the program arity. $\square$

Since Datalog LIT is a fragment of stratified Datalog, the known limitations of stratified Datalog also apply to Datalog LIT, cf. section 6.

4.3 Linear time model checking for Datalog LITE

The linear time evaluation of Datalog LIT can be extended to Datalog LITE, which is obtained from Datalog LIT by introducing a new kind of literal:

Definition 4.5. A *generalized literal* G is an expression of the form $(\forall y_1 \cdots y_n.\alpha)\beta$ where α and β are atoms, and $\mathrm{free}(\alpha) \supseteq \mathrm{free}(\beta)$. The free variables $\mathrm{free}(G)$ of G are given by $\mathrm{free}(\alpha) - \{y_1, \ldots, y_n\}$.

The intended meaning of $(\forall y_1 \cdots y_n.\alpha)\beta$ is its standard first order semantics, that is, $\forall y_1 \cdots y_n(\alpha \rightarrow \beta)$. Therefore, when the generalized literal is used within a program Π, the notion of stratification has to be adapted in such a way that for $(\forall \bar{y}.R\bar{x}\bar{y})S\bar{x}\bar{y}$, the occurrences of R are negative, and the occurrences of S are positive. In a stratified program, R must therefore belong to a lower stratum while S can be from the current stratum, too. Since the notion of free variables is well-defined for generalized literals, Definition 4.1 is applicable to rules with generalized literals.

Definition 4.6. A Datalog LITE program is a Datalog LIT program containing (unnegated) generalized literals where the notions of guardedness and stratification are extended to generalized literals as described above.

Remark 4.7. Note that the obvious translation that replaces the universal quantifier by double negation cannot be applied to recursive rules because it introduces a stratification violation. This can be seen, e.g. for the rule $Wx \leftarrow (\forall z Exz)Wz$ of Example 2.2.

Formally, the (operational) semantics of Datalog LITE programs can be obtained from the standard semantics of Datalog as follows.

The program is evaluated bottom-up stratum by stratum. After the evaluation of each stratum, the predicate values of all predicates at this stratum and at lower strata are fixed (i.e., their interpretation is fixed). The evaluation proceeds exactly as for stratified Datalog programs except that every generalized literal $L(\bar{y}) := (\forall \bar{x} \,.\, \alpha(\bar{x}, \bar{y}))\beta(\bar{x}, \bar{y})$ is instantiated (for any tuple $\bar{d}$ interpreting $\bar{y}$) by the conjunction

$$\bigwedge_{\bar{c}:\ \alpha(\bar{c},\bar{d}) \text{ true}} \beta(\bar{c}, \bar{d}).$$

Note that not all variables $\bar{x}, \bar{y}$ need actually appear in β (see the following example). Note further that the interpretation of α is defined at a lower stratum than the rule where the generalized literal appears, so there are no problems with the stratified evaluation. Just as for regular stratified Datalog, the result of a Datalog LITE program is independent of any particular chosen stratification.

Datalog LITE is indeed stronger than Datalog LIT. We illustrate this by a simple Datalog LITE program for the GAME problem. It is known that there exists no stratified Datalog program for GAME, hence in particular no Datalog LIT program.

EXAMPLE 4.8. *An instance of the GAME problem is a directed graph $G = (V, E)$ with a distinguished node v. Two players, I and II, take alternating moves, Player I begins. At each move, the players move a pebble from its current position along some edge to a new position. Initially, the pebble is at node v. Who gets stuck, looses (the opponent wins). The question to be resolved is whether Player I has a winning strategy. It is known [Abiteboul et al. 1995] that the GAME problem is P-complete via quantifier-free reductions. It is definable in the μ-calculus, by the formula $\mu X \,.\, \Diamond\Box X$, but it is not definable by a stratified Datalog query [Dahlhaus 1987; Kolaitis 1990]. However, the Datalog LITE program consisting of the single rule*

$$Wx \leftarrow Exy, (\forall z.Eyz)Wz$$

defines the set of nodes from which Player I has a winning strategy. Note that this rule is indeed guarded by the atom Exy since only y occurs free in the generalized literal $(\forall z.Eyz)Wz$.

In a non-recursive query (Π, W), universal quantification can be rewritten by double negation, obtaining an equivalent query $(\Pi^{\neg\neg}, W)$. For example, the single rule query $(\Pi, W) : Wx \leftarrow (\forall z.Exz)Mz$ is rewritten by the query $(\Pi^{\neg\neg}, W)$ which contains the two rules $Wx \leftarrow \neg W^{\neg}x$ and $W^{\neg}x \leftarrow Exz, \neg Mz$. For recursive Π however, $\Pi^{\neg\neg}$ will in general not be stratified. The well-founded semantics [Gelder et al. 1991] though assigns the correct meaning to the query:

THEOREM 4.9. *The rewriting operation $\Pi \mapsto \Pi^{\neg\neg}$ is a conservative embedding of Datalog LITE into well-founded Datalog.*

Thus, up to trivial rewriting operations, Datalog LITE is a fragment of well-founded Datalog; we have thus identified a linear time computable fragment of well-founded Datalog.

THEOREM 4.10. *The time complexity of model checking (i.e., evaluation) of Datalog LITE over finite structures is*

(a) linear in the input size and the program size for programs where all guards are input relations.

(b) linear in the input size and the program size for bounded arity programs.

(c) linear in the input size and exponential in the program size for arbitrary programs.

PROOF. (a) We shall adapt the proof of Theorem 4.4 to handle generalized literals. Over a finite input structure $\mathfrak{A}$, a generalized literal $\varphi(\bar{y}) := (\forall \bar{x}.\alpha\bar{x}\bar{y})\beta\bar{x}\bar{y}$ can for a fixed $\bar{y} = \bar{d}$ be instantiated by a finite conjunction $\bigwedge_{\bar{c}:\bar{c}\bar{d}\in\alpha^{\mathfrak{A}}} \beta\bar{c}\bar{d}$. If for some $\bar{d}$, the set $\{\bar{c} : \bar{c}\bar{d} \in \alpha^{\mathfrak{A}}\}$ is empty, the conjunction is equal to true. Thus, we can apply the following strategy: For every generalized literal $\varphi(\bar{y}) = (\forall \bar{x}.\alpha\bar{x}\bar{y})\beta\bar{x}\bar{y}$, we introduce a new relation symbol $R_\varphi(\bar{y})$, and replace all occurrences of φ by R_φ. In addition, we add ground rules with head predicate R_φ to the program. Thus, **EvalLIT** is extended as follows.

Algorithm EvalLITE$(\Pi, \mathfrak{A})$

store $\mathfrak{A}$ **in array** A
for all generalized literals $\varphi(\bar{y}) = (\forall \bar{x}.\alpha\bar{x}\bar{y})\beta\bar{x}\bar{y}$ **in** Π_i
  add the rule $R_\varphi \leftarrow$ **to HCB** H_0
  for all tuples $\bar{c}\bar{d} \in \alpha^A$
    add $\beta\bar{c}\bar{d}$ **to the body of the rule for** R_φ
for all rules $r \in \Pi$
  replace generalized literals φ **in** r **by** R_φ
call EvalLIT$(\Pi, \mathfrak{A})$

Note that **EvalLITE** uses the HCB H_0 of **EvalLIT**. Since the size of $\alpha^{\mathfrak{A}}$ is bounded by $c|\mathfrak{A}|$ (cf. the proof of Theorem 4.4), the new loop increases the size of the propositional program only linearly. It is easy to see that on a RAM, the program transformation can be done in linear time. As in the proof of Theorem 4.4, cases (b) and (c) are deferred to Corollary 4.15. ☐

Remark 4.11. Note that according to the most widely used RAM model our algorithms **EvalLIT** and **EvalLITE** use linear space because only those registers (array positions) which are effectively used are taken into account and because initialization is —as mentioned— not necessary. In the case where relation symbols are at most binary, adjacency lists of linear size can be used as data structures. In this case the space complexity is linear even when measured according to less liberal RAM models where the index of the largest used register determines space usage.

4.4 Locality invariance and input guards

Definition 4.12. Given a structure $\mathfrak{A}$ with universe A, we define the binary relation $E^{\mathfrak{A}}$ on $\mathfrak{A}$ by

$$E^{\mathfrak{A}} = \{(a, b) : \text{there is a tuple in a relation in } \mathfrak{A} \text{ containing both } a \text{ and } b\}.$$

The graph $G(\mathfrak{A}) := (A, E^{\mathfrak{A}})$ is called the *Gaifman graph* of $\mathfrak{A}$. The *distance* of two elements a and b is their distance in the Gaifman graph.

Thus, for Kripke structures, the distance coincides with the natural distance on graphs when we disregard the orientation of the edges. Note that each tuple of an input relation corresponds to a clique in the Gaifman graph. The following lemma is an immediate observation.

LEMMA 4.13. *Let Π be a Datalog LITE program with head predicates $T_1, \dots, T_r$, and let $T_1^{\mathfrak{A}}, \dots, T_r^{\mathfrak{A}}$ be the relations computed by Π on an input structure $\mathfrak{A}$. Then the Gaifman graph of the expanded structure $(\mathfrak{A}, T_1^{\mathfrak{A}}, \dots, T_r^{\mathfrak{A}})$ coincides with the Gaifman graph of $\mathfrak{A}$.*

PROOF. By induction over the strata $\Pi_0, \dots, \Pi_n$ of Π. All rules in stratum Π_0 are guarded by input relations. Thus, each tuple $(d_1, \dots, d_k)$ in a guard relation corresponds to a clique $\{d_1, \dots, d_k\}$ in the Gaifman graph of $\mathfrak{A}$. Let us fix a rule with head relation H and guard relation G. A given tuple $(d_1, \dots, d_k)$ of G can contribute only such tuples to H, which are composed of $d_1, \dots, d_k$, and thus, the contribution of the rule to the Gaifman graph can be only a subgraph of the clique $\{d_1, \dots, d_k\}$. Therefore, the Gaifman graph of the head predicates of stratum Π_0 is contained in the Gaifman graph of $\mathfrak{A}$. The induction step works analogously. $\square$

Thus, the new relations computed by a Datalog LITE program do not change the Gaifman graph of the input structure. We shall refer to this property of programs as the *locality invariance* of Datalog LITE. From this observation, one easily obtains non-expressibility results: for example, transitive closures cannot be expressed in Datalog LITE.

THEOREM 4.14. *Every Datalog LIT and Datalog LITE program is equivalent to a program where all guards are input atoms.*

PROOF. Recall that all guards are either input relations or head relations. From the locality invariance of Datalog LITE (Lemma 4.13) we conclude that all tuples appearing in guard relations appear also, possibly permuted and extended, in some input relations. Hence every rule r: $head \leftarrow body$ of Π is equivalent to the finite collection $r_1, \dots, r_l$ of all rules

$$r_i : \ head\sigma \leftarrow P\sigma, body\sigma$$

where (i) P is an atomic input predicate containing new variables, and (ii) σ is a unifier such that r_i is guarded by $P\sigma$, i.e., $P\sigma$ contains all variables of $head\sigma$ and $body\sigma$. Since the tuples in the input relations span the whole Gaifman graph, all possibilities for tuples computed by the original rule r are exhausted by the new rules. This proves the theorem for Datalog LIT.

For the case of Datalog LITE we also have to consider guards of generalized literals of form $(\forall \bar{y} \,.\, G\bar{x}\bar{y})\beta(\bar{x}, \bar{y})$. If G is an input predicate, nothing needs to be shown. Otherwise G is a head predicate of a lower stratum than the one in which the generalized literal appears. By the argument above we can assume that the rules defining G have the form

$$G\bar{z} \leftarrow \gamma_i(\bar{z}, \bar{w}), \beta_i(\bar{z}, \bar{w})$$

for $i = 1, \dots, m$, where γ_i are input guards and $\bar{z}$ is a tuple of variables of the same length as $\bar{x}\bar{y}$.

Add to the program the rules

$$L_i\bar{x}\bar{y} \leftarrow \gamma_i(\bar{x},\bar{y},\bar{w}), \neg G\bar{x}\bar{y}$$
$$L_i\bar{x}\bar{y} \leftarrow \gamma_i(\bar{x},\bar{y},\bar{w}), \beta(\bar{x},\bar{y})$$

for $i = 1, \ldots, m$ and new predicates $L_1, \ldots, L_m$. Now replace the generalized literal $(\forall\bar{y} \,.\, G\bar{x}\bar{y})\beta(\bar{x},\bar{y})$ by

$$(\forall\bar{y}\bar{w} \,.\, \gamma_1(\bar{x},\bar{y},\bar{w}))L_1\bar{x}\bar{y}, \ldots, (\forall\bar{y}\bar{w} \,.\, \gamma_m(\bar{x},\bar{y},\bar{w}))L_m\bar{x}\bar{y}$$

It is easy to verify that the new program is equivalent to the original one. By repeating this argument, each program is transformed into an equivalent that uses only input atoms as guards. □

COROLLARY 4.15. *In Datalog LIT and Datalog LITE, the time complexity of eliminating non-input guards is*

(a) *linear in the program size if the program has bounded arity.*

(b) *exponential in the program size if the program has unbounded arity.*

Thus, cases (b) and (c) of Theorems 4.4 and 4.10 are settled.

PROOF. The algorithm is immediately obtained from the proof. It is easily checked that the algorithm is exponential in the program arity, as exemplified in the following remark. □

Remark 4.16. While the use of head predicates as guards does not increase the expressive power of Datalog LITE, it permits sometimes to write programs in a much more compact way. For instance, let $\sigma_i(x_1 \cdots x_n)$ be the substitution that interchanges the variables x_i and x_{i+1} and leaves all other variables unchanged, and consider the program

$$Rx_1 \cdots x_n \leftarrow Sx_1 \cdots x_n$$
$$Rx_1 \cdots x_n \leftarrow R\sigma_i(x_1 \cdots x_n) \qquad \text{for } i = 1, \ldots, n-1.$$

This program has n rules, but the shortest equivalent program that uses only input atoms as guards has $n!$ rules.

The following proposition uses a similar argument to show that the exponential time bound is indeed optimal:

PROPOSITION 4.17. *With respect to data complexity, Datalog LIT and Datalog LITE are* PTIME*-complete. With respect to program complexity, they are* EXPTIME*-complete for unbounded arity, and remain* PTIME*-complete for bounded arity.*

PROOF. Since Datalog LITE is obviously contained in fixed point logic whose data complexity is PTIME, membership follows for both formalisms. For hardness, consider the following Datalog LIT program which solves the well-known PTIME-complete problem *Monotone Circuit Value*. Here, the monotone circuit is represented by two ternary relations $\sqcup xyz$ and $\sqcap xyz$ which denote that gate x is computed as the disjunction (conjunction resp.) of gates y and z. Moreover, three unary input relations Hx, Lx, and Ox denote high and low input gates, and the output gate respectively. Then the following query (Π, M) solves the problem:

$$
\begin{aligned}
Tx &\leftarrow Hx \\
Tx &\leftarrow \sqcup\, xyz, Tz \\
Tx &\leftarrow \sqcup\, xyz, Ty \\
Tx &\leftarrow \sqcap\, xyz, Ty, Tz \\
M &\leftarrow Ox, Tx
\end{aligned}
$$

For program complexity, the result for the bounded case follows immediately from the fact that evaluating ground Horn clause problems is PTIME-complete. Since this language is a trivial fragment of Datalog LIT, hardness follows. Membership is a trivial consequence of Theorem 4.4.

For general program complexity, it is again sufficient to show hardness for Datalog LIT. It is well-known [Gottlob et al. 1999; Vardi 1982] that the program complexity for basic Datalog programs is EXPTIME-complete even for the input database $\mathcal{K} = (\{a, b\}, E^{\mathcal{K}})$, where $E^{\mathcal{K}} = \{(a, b)\}$.

We reduce this problem to program complexity of Datalog LIT. Given a program Π over domain $\mathcal{K}$, we construct an equivalent Datalog LIT program Π'. Let n be the maximal arity of relations in Π. We define additional relations $D_1, D_2, \dots D_n$ (where each D_i has arity $i + 2$) by the following program Π:

$$
\begin{aligned}
D_1xxy &\leftarrow Exy \\
D_1yxy &\leftarrow Exy \\
D_2xzxy &\leftarrow D_1zxy \\
D_2yzxy &\leftarrow D_1zxy \\
&\vdots \\
&\vdots \\
D_ix\bar{z}xy &\leftarrow D_{i-1}\bar{z}xy \\
D_iy\bar{z}xy &\leftarrow D_{i-1}\bar{z}xy \\
&\vdots \\
D_nx\bar{z}xy &\leftarrow D_{n-1}\bar{z}xy \\
D_ny\bar{z}xy &\leftarrow D_{n-1}\bar{z}xy
\end{aligned}
$$

Thus, D_j contains all tuples of the form $\{a, b\}^j ab$. Let $r : \mathit{head} \leftarrow \mathit{body}$ be a rule in Π containing variables $x_1, \dots, x_j$, $j \leq n$. Since all relations in r range only over a, b, the rule r is equivalent to the guarded rule $r' : \mathit{head} \leftarrow D_j x_1 \dots x_j xy, \mathit{body}$. The program Π' is obtained by the rules for the D_i and the new guarded rules of the form r'. Since Π' can be computed in polynomial time, the result follows. □

5. THE EXPRESSIVE POWER OF DATALOG LIT AND DATALOG LITE OVER KRIPKE STRUCTURES

LEMMA 5.1. *Over Kripke structures, every Datalog LITE program is equivalent to a program containing only predicates of arity at most* 2.

PROOF. By Theorem 4.14 we can assume without loss of generality that only input predicates are used as guards. Hence each rule contains at most two free variables x, y. Therefore, if the program contains a head predicate K of arity k, all its occurrences are of the form Kw where $w \in \{x, y\}^k$. The proof idea is to introduce 2^k new binary predicates K_w which correspond to the specific tuple pattern. We call such a w a variable pattern. The program transformation is then obtained by replacing all occurrences of a literal Kw either in the body or the head of a rule by $K_w xy$. Variable patterns which are obtained by interchanging the variables x and y, have to be identified. Therefore, we add the rules $K_w xy \leftarrow K_{w'} yx$ for all tuples w, w' where w' is the tuple obtained from interchanging x and y. Moreover, the rule $K_w xx \leftarrow K_{x^k} xy$ reflects that x and y in the variable patterns may be equal. An easy, yet somewhat tedious, inductive argument shows that this translation indeed yields a correct program. □

LEMMA 5.2. *Over Kripke structures, every Datalog LITE program is equivalent to a Datalog LITE program where the binary predicates all occur with two different variables.*

PROOF. For every binary relation symbol F, we introduce a new relation symbol F_{xx}. For every rule in the program, we add a new rule where all variables are equal. Then all occurrences of literals of the form Fzz in the resulting program are replaced by $F_{xx}z$. □

LEMMA 5.3. *Over Kripke structures, every Datalog LIT program with monadic output predicate is equivalent to a Datalog LIT program where all head predicates are monadic.*

PROOF. We will show that every binary relation F which occurs in a program Π, can be defined by a finite number of rules which contain only input relations and monadic relations. Moreover, a similar definition can be found for the complement of F. The idea of the proof is that the binary relations can be expanded by using their defining rules. The guards guarantee that no new variables can be introduced, and thus, the expansion terminates after a finite number of steps. By Lemma 5.2 we may assume, without loss of generality, that no binary relation symbol B occurs in the form Bxx.

Let us consider a relation R. To *expand* a defining rule r for R means to systematically replace the binary literals $(\neg)L$ in the body of r by conjunctions which are obtained from the bodies of those rules which have L as a head predicate; thus a set of new rules is obtained which replaces the original rule.

For positive literals this is easy. The binary relation symbol is replaced by the body of the rule, and variables are possibly renamed. (Note that we may w.l.o.g. assume that just two free variables x, y are used in the whole program; thus, during expansion, either the body of a rule is copied, or it is copied with interchanged variable names.) Since guards appear only as positive literals, they are replaced only by rule bodies of guarded rules; thus every guard is replaced by a conjunction which again contains a guard, and therefore, guardedness is preserved.

For negated literals we introduce a new notion: Given a binary relation F, a *failure justification* is a conjunction γ of literals which is obtained by choosing one literal from the body of every rule with head F, and negating it. Since a negative literal $\neg L$ cannot be a guard anyway, $\neg L$ can be safely replaced by all possible γ's without affecting the guardedness of the rule.

For recursion-free programs, this process trivially terminates, since the dependency graph of the program is a tree. In the general case, consider the dependency graph D of the program restricted to binary relations. Recall that the program is stratified, i.e., there are no cycles in the dependency graph contain negations. Suppose that there is a cycle containing a relation symbol F. Then any path from F to F (i.e., any cycle containing F) corresponds to one or more expansions of a rule for F which eventually leads to F again.

Because of guardedness there are only two possibilities: either a rule of the form $Fxy \leftarrow Fyx, \ldots$ is obtained, or a rule of the form $Fxy \leftarrow Fxy, \ldots$. A rule of the first form is fine, and can be further expanded, but a rule of the second form is useless since it requires Fxy to derive Fxy. Therefore, it is redundant in the program and can be removed.

Let us continue with the case of $Fxy \leftarrow Fyx, \ldots$. If the further expansion of Fyx leads either to Fxy or to Fxy, the resulting rule will be removed since it is redundant. In conclusion we see that every node in the dependency graph has to be visited at most twice. Therefore, the expansion terminates.

We conclude that there is no recursion between binary relations except for the very restricted form $Fxy \leftarrow Fyx, \ldots$ which can be easily eliminated. Since the output predicate is monadic, it follows that all binary head predicates can be eliminated from the program by systematic expansion. $\square$

THEOREM 5.4. *Datalog LITE with only unary and binary input predicates is contained in monadic second order logic.*

PROOF. (Sketch) An argument similar to the proof of Theorem 5.3 shows that while in general binary predicates cannot be removed from Datalog LITE programs – consider a case like $(\forall x.\alpha)Bxy$ where B is defined by $Bxy \leftarrow Exy$ and $Bxy \leftarrow Fxy$ – there is no recursion via binary predicates. Hence, there is a translation into monadic fixed point logic, and thus, into monadic second order logic. $\square$

By the Theorem of Janin and Walukiewicz [Janin and Walukiewicz 1996] that on Kripke structures, every property that is definable in monadic second-order logic and invariant under bisimulation, is definable also in the μ-calculus, we obtain the following consequence.

COROLLARY 5.5. *Every monadic, bisimulation-invariant Datalog LITE query over Kripke structures is definable in the μ-calculus.*

In Section 7, we introduce *modal* Datalog LITE. Modality is a sufficient syntactic criterion for bisimulation-invariance.

6. LIMITS OF STRATIFIED DATALOG

In this section, we substantiate our claims of Section 3.2 about the insufficiency of stratified Datalog for expressing temporal properties.

Well-foundedness statements are expressed in CTL by formulae of the form $\mathbf{AF}\varphi$ and in the μ-calculus by $\mu X.\varphi \vee \Box X$, where φ is any (non-contradictory) formula (for instance, an atomic proposition). The intuitive reason why, over infinite structures, such statements are not expressible by stratified programs is that they require recursion through a universal statement, whereas Datalog (even stratified Datalog) has only recursion through existential statements.

We establish a more general result, in terms of an appropriate fragment of infinitary logic.

Definition 6.1. The logic $L_{\omega_1\omega}$ extends first-order logic by the possibility to take disjunctions $\bigvee \Phi$ and conjunctions $\bigwedge \Phi$ over countable sets Φ of formulae. Inside this logic we define a hierarchy of fragments $L(\Sigma_k)$ and $L(\Pi_k)$ for $k < \omega$. The bottom level $L(\Sigma_0) = L(\Pi_0)$ is the set of quantifier-free formulae in $L_{\omega_1\omega}$. Further, for every k, $L(\Pi_k) := \{\neg\varphi : \varphi \in L(\Sigma_k)\}$ and $L(\Sigma_{k+1})$ is the class of formulae $\bigvee \Phi$ where Φ is a countable subset of $\{\exists x_1 \cdots \exists x_m \varphi : m < \omega, \varphi \in L(\Pi_k)\}$.

LEMMA 6.2. *Every query in stratified Datalog is equivalent to some infinitary formula in* $\bigcup_{k<\omega} L(\Sigma_k)$.

The proof is a straightforward induction over the number of strata in a stratified program.

THEOREM 6.3. *Over countable structures, well-foundedness statements are not expressible in* $\bigcup_{k<\omega} L(\Sigma_k)$. *In particular they are not expressible by stratified Datalog programs.*

PROOF. The proof is inspired by a proof in [Flum 1999]. For each ordinal α, we define a directed tree T_α, with edge relation E and a unary relation P, inductively as follows. T_0 consists of a single element, the root of T_0. For $\alpha > 0$, T_α is constructed by taking the disjoint union of all trees T_β for $\beta < \alpha$, and adding a new element, the root of T_α, which is connected by an edge to the root of each T_β, for $\beta < \alpha$. On each T_α, P consists of the leaves, i.e. the nodes that do not have an E-successor.

Further, let T_α^+ be T_α together with an edge from the root to itself, and let S_α be the unfolding of T_α^+ as a tree. Note that S_α can be viewed as $T_\alpha \times \omega$ where there is an edge from (t, i) to (t', j) if and only if either $i = j$ and t' is a child of t in T_α, or $j = i + 1$ and $t = t' = \mathit{root}$.

Let ψ be the well-foundedness statement $\mu X.P \vee \Box X$, saying that on each path one will eventually reach P. Clearly, for each α, T_α is well-founded, but S_α is not. To put it differently, $(T_\alpha, \mathit{root}) \models \psi$ but $(S_\alpha, \mathit{root}) \models \neg\psi$. We want to show that ψ is not equivalent to any formula in $\bigcup_{k<\omega} L(\Sigma_k)$. It suffices to establish the following result.

Claim. For all $k < \omega$, $T_{k\omega} \equiv_{L(\Sigma_k)} S_{k\omega}$

A straightforward adaption of the Ehrenfeucht-Fraïssé method for infinitary languages shows that two structures $\mathfrak{A}$ and $\mathfrak{B}$ are $L(\Sigma_k)$-equivalent if there exists a sequence $(I_j)_{j\le k}$ of non-empty sets of finite local isomorphisms from $\mathfrak{A}$ to $\mathfrak{B}$ with the following back and forth property.

Forth: . If $p \in I_{j+1}$, $m < \omega$ and $a_1, \ldots, a_m$ in $\mathfrak{A}$, then there exist $b_1, \ldots, b_m$ in $\mathfrak{B}$ and $q \in I_j$ such that $q \supseteq p \cup \{(a_1, b_1), \ldots, (a_m, b_m)\}$.

Back: . If $p \in I_{j+1}$, $m < \omega$ and $b_1, \ldots, b_m$ in $\mathfrak{B}$, then there exist $a_1, \ldots, a_m$ in $\mathfrak{B}$ and $q \in I_j$ such that $q \supseteq p \cup \{(a_1, b_1), \ldots, (a_m, b_m)\}$.

For u, v in T_α or S_α we write $u < v$ if u is a predecessor of v, i.e. if u lies on the path from the root to v. The height of an element u, denoted $h(u)$ is 0 if u is a leaf and $\sup\{1 + h(v) : (u, v) \in E\}$ otherwise. For each $j \le k$, let I_j be the set of local isomorphism p from $T_{k\omega}$ to $S_{k\omega}$ such that:

(a) p is closed under predecessors: If $(a, b) \in p$ then for all $a' < a$ (and all $b' < b$) there exists a $b' < b$ (an $a' < a$) such that $(a', b') \in p$.

(b) If $(a,b) \in p$ and $h(a) < j\omega$, then $h(b) = h(a)$.
(c) If $(a,b) \in p$ and $h(a) \geq j\omega$, then $h(b) \geq j\omega$.

It is not difficult to verify that $(I_j)_{j \leq k}$ satisfies the back and forth properties mentioned above. $\square$

LEMMA 6.4. *Even over finite structures, well-foundedness statements cannot be expressed by stratified Datalog programs with only monadic head predicates.*

PROOF. Let Π be a stratified Datalog program with a single binary input predicate E and monadic head predicates $H_1, \ldots, H_r$. Then there exist formulae $\chi_{H_i}(x)$ in monadic second-order logic (MSO) which are equivalent to the queries (Π, H_i). Suppose, towards a contradiction, that (Π, H_1) computes the well-founded elements of the input graph, that is, the set of nodes v such that the graph contains no infinite E-path originating in v.

For each n, let S_n be the graph $(\{0, \ldots, n\}, \{(j, j+1) : 0 \leq j < n\})$ (i.e. the path of length n). The evaluation of Π on S_n leads to the word structure $W_n = (S_n, \chi_{H_1}^{S_n}, \ldots, \chi_{H_r}^{S_n})$ of vocabulary $\tau := (E, H_1, \ldots, H_r)$. For two ordered structures $\mathfrak{A}$ and $\mathfrak{B}$, let $\mathfrak{A} \lhd \mathfrak{B}$ denote the ordered sum of $\mathfrak{A}$ and $\mathfrak{B}$, cf. [Ebbinghaus and Flum 1999]. (Note that $\lhd$ is associative.) Moreover, let $\mathfrak{A}^k$ denote the $k-$fold ordered sum of $\mathfrak{A}$.

Let L be the language $\{W_n : n \in \mathbb{N}\}$, and ρ the MSO-sentence $\bigwedge_{1 \leq i \leq h}(\forall x. \chi_{H_i}(x) \leftrightarrow H_i x)$. Then for every word structure W of vocabulary τ, $W \models \rho$ if and only if W is isomorphic to some $W_i \in L$. Hence, by Büchi's Theorem, L describes a regular language. Since L is infinite, the Pumping Lemma implies that there exist word structures W_i, W_j, W_k, such that for all $\ell \geq 1$, $W_i \lhd W_j^\ell \lhd W_k \in L$. Note that we can without loss of generality choose i, j and k arbitrarily large.

We conclude that for sufficiently large i, j, k the Datalog program Π cannot distinguish input structures of the form $S_{i+\ell j+k}$ from the root. Now consider the structure $\mathfrak{A} = S_i \lhd S_j \lhd S_j \lhd S_k$. In $\mathfrak{A}$, let c be the node connecting the two copies of S_j, and obtain a second structure $\mathfrak{B}$ by adding a cycle of length j to the node c.

Obviously, the root of $\mathfrak{A}$ is well-founded, while the root of $\mathfrak{B}$ is not. We claim that Π does not distinguish $\mathfrak{A}$ and $\mathfrak{B}$ on their common elements. To see this, we consider the contribution of the strata $\Pi_1, \ldots, \Pi_p$ of Π. For simplicity, let us identify relations on the graphs with colors, and let m be the number of variables of Π.

We show that after evaluation of each stratum of Π, the common elements of $\mathfrak{A}$ and $\mathfrak{B}$ are colored identically, and the j-cycle in $\mathfrak{B}$ is colored isomorphically to the copies of S_j in $\mathfrak{A}$ and $\mathfrak{B}$. Suppose that this condition is satisfied before the evaluation of stratum Π_i (clearly this is the case for $i = 1$).

Each stratum corresponds to an existential least fixed point formula, and every Datalog rule expresses an existential first-order statement which is positive in the new colors of that stratum. By induction hypothesis the m-neighborhoods around corresponding elements of $\mathfrak{A}$ and $\mathfrak{B}$ satisfy the same existential statements concerning the old colors. Therefore, for sufficiently large i, j, k, the loop and the copies of S_j are colored isomorphically. Operationally, one can imagine that first the substructure $\mathfrak{A}$ of $\mathfrak{B}$ is colored, and then the loop has to be colored identically, since the existential statements are true on the loop if they were true on S_j. $\square$

Lemma 5.3 together with Lemma 6.4 imply that Datalog LIT does not express all of CTL.

THEOREM 6.5. *Datalog LIT and CTL are semantically incomparable.*

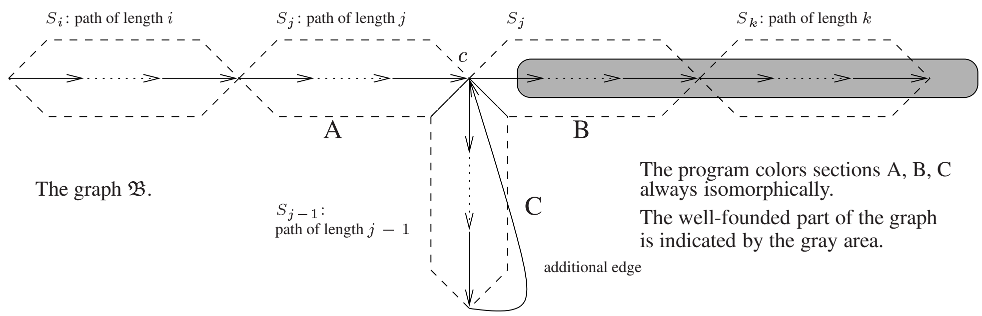

Fig. 2. Illustration of Lemma 6.4.

7. CAPTURING CTL AND THE ALTERNATION-FREE μ-CALCULUS

In this section we show that three common temporal verification formalisms, namely propositional (multi-)modal logic ML, CTL, and the alternation-free portion of the μ-calculus can be captured by appropriate fragments of Datalog LITE.

Definition 7.1. A *modal* Datalog-program Π is a Datalog LITE program such that

(1) All input predicates of Π are unary or binary and all head predicates are unary.
(2) All rules have one of the following forms:

$$Hx \leftarrow \mathrm{mon}(x)$$
$$Hx \leftarrow \mathrm{mon}(x), Exy, \mathrm{mon}(y)$$
$$Hx \leftarrow \mathrm{mon}(x), (\forall y.Exy)Py$$

where $\mathrm{mon}(z)$ may be any sequence $(\neg)P_1z, \ldots, (\neg)P_rz$ of monadic literals in the variable z.

Remark 7.2. Note that rules of the form $Hx \leftarrow \mathrm{mon}(x), (\forall y.Exy)Py$ are monadic, since the generalized literal contains only one free variable x.

The justification for calling these programs modal is given by the following observation.

LEMMA 7.3. *Every query defined by a modal Datalog program is invariant under bisimulation.*

It is easy to see that every modal program is equivalent to a *reduced modal program* where all rules are of the form

$$Hx \leftarrow (\neg)Px, (\neg)Qx \qquad Hx \leftarrow Exy, Py \qquad Hx \leftarrow (\forall y\,.\,Exy)Py.$$

Note that P, Q and H need not be distinct.

EXAMPLE 7.4. *The rule*

$$r : Hx \leftarrow Ax, Exy, Cy, \neg Dy$$

is equivalent to the rules

$$r' : Hx \leftarrow Ax, M_1x; \qquad M_1x \leftarrow Exy, M_2y; \qquad M_2x \leftarrow Cx, \neg Dx.$$

We say that a modal logic L, like e.g. ML, CTL or the μ-calculus, is equivalent to a class $\mathcal{C}$ of modal Datalog programs if for every Datalog query (Π, H) with $\Pi \in \mathcal{C}$ there exists a sentence $\psi \in L$, and vice versa, such that, for all Kripke structures $\mathcal{K}$ and all states v of $\mathcal{K}$,

$$\mathcal{K}, v \models \psi \iff v \in (\Pi, H)[\mathcal{K}].$$

Recall that a Datalog program is non-recursive if there are no circular dependencies of its predicates (for a formal definition, see [Abiteboul et al. 1995, Chapter 5.2].)

PROPOSITION 7.5. *Propositional multi-modal logic is equivalent to the class of non-recursive modal Datalog programs.*

PROOF. This is a simple observation, but we present the proof in detail since it is the basis for the further equivalence results following below.

Given a non-recursive modal program Π with binary predicates E_a ($a \in A$) we construct for each monadic predicate H of Π a formula ψ_H of propositional multi-modal logic with modalities (actions) $a \in A$, such that ψ_H expresses the query (Π, H).

By renaming variables, if necessary, we can assume that all rules with head predicate H have head variable x. Further, we can assume that the program is in reduced form and, since the program is non-recursive, that all monadic predicates S in the body of a rule with head Hx are either input predicates (in which case we take $\psi_S := S$) or head predicates for which the formula ψ_S has already been constructed.

For every rule r of form $Hx \leftarrow (\neg)Px, (\neg)Qx$ let $\varphi_r := (\neg)\psi_P \wedge (\neg)\psi_Q$. For every rule of form $Hx \leftarrow Exy, Py$, let $\varphi_r := \langle a\rangle\psi_P$, and for every rule of form $Hx \leftarrow (\forall y \,.\, Exy)Py$, let $\varphi_r := [a]\psi_P x$. Finally take for ψ_H the disjunction of all the formulae φ_r for all rules with head Hx. It is easy to see that ψ_H is equivalent to (Π, H).

Conversely, we construct for every sentence $\psi \in$ ML a non-recursive modal Datalog program Π_ψ with distinguished head predicate H_ψ such that the query (Π_ψ, H_ψ) is equivalent to ψ. If ψ is an atomic proposition P, then Π_ψ consists of the single rule $H_\psi x \leftarrow Px$. If programs for the subformulae of ψ have already been constructed, then extend these programs as follows:

(1) $\psi = \varphi \wedge \vartheta$. Add the rule $H_\psi x \leftarrow H_\varphi x, H_\vartheta x$.
(2) $\psi = \varphi \vee \vartheta$. Add the rules $H_\psi x \leftarrow H_\varphi x$ and $H_\psi x \leftarrow H_\vartheta x$.
(3) $\psi = \neg\varphi$. Add a new stratum consisting of the rule $H_\psi x \leftarrow \neg H_\varphi x$.
(4) $\psi = \langle a\rangle\varphi$. Add the rule $H_\psi x = E_a xy, H_\varphi y$.
(5) $\psi = [a]\varphi$. Add the rule $H_\psi x = (\forall y \,.\, E_a xy)H_\varphi y$.

□

Since one can eliminate the □-operators in modal formulae, it follows that even non-recursive modal Datalog LIT-programs suffice to capture ML. On the other side, we can also keep the □-operators and push all negations through the modal operators so that they only appear in front of atomic propositions. The translation then gives us a basic (i.e. non-stratified) modal Datalog LITE program.

CTLog programs

Definition 7.6. A CTLog program is a modal Datalog LITE program Π satisfying the following conditions:

(1) Every head predicate appears in at most one recursive rule (i.e. a rule where the head predicate occurs also in the body of the rule). Moreover, this rule has the form

$$Hx \leftarrow Px, Exy, Hy \qquad \text{or}$$
$$Hx \leftarrow Px, (\forall y \,.\, Exy)Hy$$

(2) If we remove these rules, the remaining program is non-recursive.

PROPOSITION 7.7. *Every* CTL-*sentence* ψ *is equivalent to a query* (Π_ψ, H_ψ) *where* Π_ψ *is a CTLog-program.*

PROOF. The translation is precisely the same as in the proof of Proposition 7.5 except for CTL-sentences of form $\mathbf{E}(\varphi\mathbf{U}\vartheta)$ and $\mathbf{A}(\varphi\mathbf{U}\vartheta)$. (Recall that $\mathbf{EX}\varphi$ and $\mathbf{AX}\varphi$ are equivalent to $\Diamond\varphi$ and $\Box\varphi$, respectively.

—$\psi := \mathbf{E}(\varphi\mathbf{U}\vartheta)$. Add to the programs Π_φ and Π_ϑ the two rules

$$H_\psi x \leftarrow H_\vartheta x$$
$$H_\psi x \leftarrow H_\varphi x, Exy, H_\psi y$$

—$\psi = \mathbf{A}(\varphi\mathbf{U}\vartheta)$. Add to Π_φ and Π_ϑ two strata as follows:

$$\begin{aligned} Rx &\leftarrow H_\vartheta x \\ Rx &\leftarrow (\forall y \,.\, Exy)Ry \\ Sx &\leftarrow \neg H_\varphi x, \neg H_\vartheta x \\ Sx &\leftarrow \neg H_\vartheta x, Exy, Sy \\ H_\psi x &\leftarrow Rx, \neg Sx \end{aligned}$$

The correctness of the translation is obvious for $\mathbf{E}(\varphi\mathbf{U}\vartheta)$. To see that the program for $\psi = \mathbf{A}(\varphi\mathbf{U}\vartheta)$ is correct, note that

$$\mathbf{A}(\varphi\mathbf{U}\vartheta) \equiv \mathbf{AF}\vartheta \wedge \neg\mathbf{E}(\neg\vartheta\mathbf{U}(\neg\varphi \wedge \neg\vartheta)).$$

The first stratum computes the set R of states at which $\mathbf{AF}\vartheta$ holds and the set S of states where $\mathbf{E}(\neg\varphi\mathbf{U}(\neg\vartheta \wedge \neg\vartheta))$ is true (see case (5)). The second stratum takes the conjunction of R with the complement of S. $\Box$

The converse is also true.

PROPOSITION 7.8. *Every query computed by a CTLog-program is expressible by a* CTL-*formula.*

PROOF. Let Π be a CTLog program and H be a head predicate of Π. There is in Π at most one recursive rule with head Hx and, further, a collection $r_1, \ldots, r_m$ of rules with head Hx from the non-recursive portion of Π. The latter can be translated as in the proof of Proposition 7.5 into sentences $\varphi_{r_1}, \ldots, \varphi_{r_m}$ that are built via the basic propositional connectives and the modal operators $\Diamond$ and $\Box$ (or, in CTL-syntax, $\mathbf{EX}$ and $\mathbf{AX}$) from previously constructed CTL-sentences ψ_P where P are extensional monadic predicates or head predicates from lower strata.

Let $\varphi_H := \varphi_{r_1} \vee \cdots \vee \varphi_{r_m}$. If there is no recursive rule with head H, set $\psi_H := \varphi_H$. If the recursive rule with head H has the form $Hx \leftarrow Px, Exy, Hy$ then set

$$\psi_H := \mathbf{E}(\psi_P\mathbf{U}\varphi_H)$$

and if the recursive rule with head H has the form $Hx \leftarrow Px, (\forall y \,.\, Exy)Hy$ then set

$$\psi_H := \mathbf{A}(\psi_P \mathbf{U} \varphi_H).$$

It is easily verified that ψ_H is equivalent to (Π, H). □

PROPOSITION 7.9. *With respect to data complexity, CTLog is* NLOGSPACE-*complete. With respect to program complexity, CTLog is* PTIME-*complete*

PROOF. It is easy to see that CTLog can be translated in transitive closure logic, and thus the upper bound for data complexity follows by [Immerman 1988]. For the lower bound it is sufficient to see that the reachability query for digraphs is expressible in CTLog. Since the evaluation of non-recursive ground programs is PTIME-complete [Vardi 1982], the program complexity follows. For a detailed proof of PTIME-completeness, see also [Dantsin et al. 1997]. □

THEOREM 7.10. *The alternation-free portion of the μ-calculus is equivalent to the class of all modal Datalog programs.*

PROOF. We translate alternation-free sentences $\psi \in L_\mu$ into modal Datalog queries (Π_ψ, H_ψ). We start the procedure by eliminating the greatest fixed points, using the duality $\nu X.\varphi \equiv \neg\mu X.\neg\varphi[X/\neg X]$ and push the negation through the Boolean connectives and the modal operators so that we end up with a formula that has negation signs only in front of μ-operators and in front of atomic propositions. The fact that ψ is alternation-free means that in every subformula $\mu X.\varphi$ the fixed point variable X does not appear in the scope of a negation sign. (This will be the reason why the resulting program is stratified.)

We then proceed by induction as in the proof of Proposition 7.5. It only remains to define an appropriate translation for the formulae of the form $\mu X.\varphi$. By induction hypothesis φ is already correctly translated into a modal program Π_φ with head predicate H_φ. The variable X in φ corresponds to an extensional monadic predicate X in Π_φ with only positive occurrences. Further since X never appears in φ in the scope of a negation, and negation is the only operation that introduces a new stratum, X is only used in the top stratum of Π_φ. The program Π_ψ for $\psi := \mu X.\varphi$ is obtained by replacing in Π_φ both X and H_φ by H_ψ. There are only positive occurrences of H_ψ in Π_ψ, the predicate H_ψ is only used in the top stratum, and Π_ψ computes the least fixed point of φ as required.

For the converse, suppose that Π is a modal Datalog program with strata $(\Pi_1, \dots, \Pi_r)$. We proceed by induction over the strata, assuming that for each predicate S that is either extensional or a head predicate from a lower stratum than Π_i, we already have a formula ψ_S from the alternation-free μ-calculus which is equivalent to the query (Π, S). We can assume that the stratum Π_i contains head predicates $Z_1, \dots, Z_m$. and that all rules of Π_i with head predicate Z_i either contain in the body only extensional predicates and head predicates from lower strata, or have one of the forms

$$\begin{aligned}
Z_i x &\leftarrow Z_j x, (\neg)Px \\
Z_i x &\leftarrow Z_j x, Z_k x \\
Z_i x &\leftarrow E_a xy, Z_j y \\
Z_i x &\leftarrow (\forall y. E_a xy) Z_j y.
\end{aligned}$$

Each such rule r is translated into a formula φ_r. For rules that do not depend on any Z_i the translation is precisely as in the proof of Proposition 7.5.

—If r has the form $Z_i x \leftarrow Z_j x, (\neg)Px$, then $\varphi_r := Z_j \wedge (\neg)\psi_P$,

—if r is $Z_i x \leftarrow Z_j x, Z_k x$, then $\varphi_r := Z_j \wedge Z_k$,

—if r is $Z_i x \leftarrow E_a xy, Z_j y$, $\varphi_r := \langle a \rangle Z_j$, and finally

—if r is $Z_i x \leftarrow (\forall y.E_a xy) Z_j y$, then $\varphi_r := [a] Z_j$.

For each head predicate Z_i we obtain a formula $\eta_i(Z_1, \dots, Z_r)$ by taking the disjunction the formulae φ_r for all rules with head predicate Z_i. The result that Π computes for $Z_1, \dots, Z_m$ is the simultaneous least fixed point $(Z_1^\infty, \dots, Z_m^\infty)$ of the operator defined by

$$\eta_1(Z_1, \dots, Z_m), \dots, \eta_m(Z_1, \dots, Z_m).$$

By the Transitivity Lemma on inductive definitions (see e.g. [Ebbinghaus and Flum 1999, p. 180]) each of $Z_1^\infty, \dots, Z_m^\infty$ can equivalently be defined as a nested least fixed point. For $m = 2$, for instance, Z_1^∞ and Z_2^∞ are the nested fixed point defined, respectively, by the μ-formulae

$$\psi_{Z_1} := \mu Z_1 \,.\, \eta_1(Z_1, \mu Z_2 \,.\, \eta_2(Z_1, Z_2))$$
$$\psi_{Z_2} := \mu Z_2 \,.\, \eta_2(\mu Z_1 \,.\, \eta_1(Z_1, Z_2), Z_2).$$

This generalizes to arbitrary m and gives us for each head predicate Z of Π the desired formula ψ_Z. □

PROPOSITION 7.11. *Modal Datalog is* PTIME*-complete with respect to data complexity.*

8. DATALOG LITE AND GUARDED FIXED POINT LOGIC

In this section we show that Datalog LITE is equivalent to a natural fragment of the guarded fixed point logic μGF which has recently been studied by Grädel and Walukiewicz [Grädel and Walukiewicz 1999]. The idea to consider *guarded fragments* of first-order logic and its extensions is due to Andréka, van Benthem and Németi [Andréka et al. 1998]. Their main goal was to identify the reasons for the convenient model-theoretic and algorithmic properties of modal logics and to and generalize the modal fragment of first-order logic (obtained by the standard embedding of propositional modal formulae into first-order logic) to more expressive fragments that nevertheless retain the good properties of modal logics. Starting from the observation that in the standard translation from modal logic the quantifiers are used only in a very restricted way, they defined the *guarded fragment* of first-order logic. They dropped the restrictions of the modal fragment that only two variables and only monadic and binary predicates may be used, but imposed instead that all quantifiers must be appropriately relativized by atomic formulae. For an informal discussion of various guarded logics, see [Grädel 1999c].

Definition 8.1. The *guarded fragment* GF of first-order logic is defined inductively as follows.

(1) Every relational atomic formula belongs to GF.

(2) GF is closed under propositional connectives.

(3) If $\bar{x}, \bar{y}$ are tuples of variables, $\alpha(\bar{x}, \bar{y})$ is a positive atomic formula and $\psi(\bar{x}, \bar{y})$ is a formula in GF such that $\mathrm{free}(\psi) \subseteq \mathrm{free}(\alpha) = \bar{x} \cup \bar{y}$, then the formulae

$$\exists\bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$$
$$\forall\bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x}, \bar{y}))$$

belong to GF.

Here $\mathrm{free}(\psi)$ means the set of free variables of ψ. An atom $\alpha(\bar{x}, \bar{y})$ that relativizes a quantifier as in rule (3) is the *guard* of the quantifier. Notice that the guard must contain *all* the free variables of the formula in the scope of the quantifier.

Notation. We will use the notation $(\exists\bar{y} \,.\, \alpha)$ and $(\forall\bar{y} \,.\, \alpha)$ for relativized quantifiers, i.e., we write guarded formulae in the form $(\exists\bar{y} \,.\, \alpha)\psi(\bar{x}, \bar{y})$ and $(\forall\bar{y} \,.\, \alpha)\psi(\bar{x}, \bar{y})$. When this notation is used, then it is always understood that α is indeed a proper guard as specified by condition (3).

The guarded fragment GF extends the modal fragment and turns out to have interesting properties [Andréka et al. 1998; Grädel 1999b]: (1) The satisfiability problem for GF is decidable; (2) GF has the finite model property, i.e., every satisfiable formula in the guarded fragment has a finite model; (3) GF has (a generalized variant of) the tree model property; (4) Many important model theoretic properties which hold for first-order logic and modal logic, but not, say, for the bounded-variable fragments FO^k, do hold also for the guarded fragment; (5) The notion of equivalence under guarded formulae can be characterized by a straightforward generalization of bisimulation.

Further, in [Grädel 1999b] it is shown that the satisfiability problem for GF is 2EXPTIME-complete in the general case and EXPTIME-complete in the case where all relation symbols have bounded arity.

The guarded fixed point logic μGF is the extension of GF by least and greatest fixed points, and it relates to GF in the same way as the μ-calculus relates to propositional modal logic and as least fixed point logic FO(LFP) relates to first-order logic.

Definition 8.2. The guarded fixed point logic $\mu\mathrm{GF}$ is obtained by adding to GF the following rules for constructing fixed-point formulae:

Let W be a k-ary relation symbol, $\bar{x} = x_1, \ldots, x_k$ a k-tuple of distinct variables and $\psi(W, \bar{x})$ be a guarded formula that contains only positive occurrences of W, no free variables other than $x_1, \ldots, x_k$ and where W is not used in guards. Then we can build the formulae

$$[\mathrm{LFP}\ W\bar{x} \,.\, \psi](\bar{x}) \qquad \text{and} \qquad [\mathrm{GFP}\ W\bar{x} \,.\, \psi](\bar{x}).$$

The semantics of the fixed point formulae is the usual one: Given a structure $\mathfrak{A}$ and a valuation χ for the free relation variables in ψ, except W, the formula $\psi(W, \bar{x})$ defines an operator on k-ary relations $W \subseteq A^k$, namely

$$\psi^{\mathfrak{A},\chi}(W) := \{\bar{a} \in A^k : \mathfrak{A}, \chi \models \psi(W, \bar{a})\}.$$

Since W occurs only positively in ψ, this operator is monotone (i.e., $W \subseteq W'$ implies $\psi^{\mathfrak{A},\chi}(W) \subseteq \psi^{\mathfrak{A},\chi}(W')$) and therefore has a least fixed point LFP($\psi^{\mathfrak{A},\chi}$) and a greatest fixed point GFP($\psi^{\mathfrak{A},\chi}$). Now, the semantics of least fixed point formulae is defined by

$$\mathfrak{A}, \chi \models [\mathrm{LFP}\ W\bar{x} \,.\, \psi(W, \bar{x})](\bar{a}) \quad \text{iff} \quad \bar{a} \in \mathrm{LFP}(\psi^{\mathfrak{A},\chi})$$

and similarly for the greatest fixed points.

It is obvious that μGF generalizes the μ-calculus. On the other side it is not difficult to see that μGF does *not* have the finite model property (see [Grädel and Walukiewicz 1999] for an example of an infinity axiom in μGF). However, μGF shares most of the other model-theoretic and algorithmic properties of the μ-calculus. In particular, μGF has the generalized tree model property and its satisfiability problem is decidable via automata-theoretic methods. Also the complexity of μGF could be identified precisely [Grädel and Walukiewicz 1999] (see also [Grädel 1999a]).

THEOREM 8.3 GRÄDEL, WALUKIEWICZ. *The satisfiability problem for guarded fixed point logic is* 2EXPTIME*-complete. For every $k \geq 2$, the satisfiability problem for μGF-sentences with relation symbols of arity at most k is* EXPTIME*-complete.*

We will now show that Datalog LITE coincides with the alternation-free part of μGF. This lifts the correspondence of modal Datalog with the alternation-free μ-calculus (see Theorem 8.5) to the first-order framework.

Definition 8.4. A sentence in μGF is *alternation-free* if it does not contain subformulae $\psi := [\mathrm{LFP}\ T\bar{x}\,.\,\varphi](\bar{x})$ and $\vartheta := [\mathrm{GFP}\ S\bar{y}\,.\,\eta](\bar{y})$ such that T occurs in η and ϑ is a subformula φ, or S occurs in φ and ψ is a subformula of η.

Equivalently, we can put this as follows. If we eliminate greatest fixed points from the given sentence (using the duality to least fixed points) and push negations through quantifiers and Boolean operators so that the resulting sentence has negations only in front of atoms and in front of LFP-operators, then no fixed-point variable must occur in the scope of a negation.

THEOREM 8.5. *Every alternation-free sentence in μGF is equivalent to a Datalog LITE query. Conversely, every Datalog LITE query is equivalent to an alternation free formula in μGF.*

PROOF. Let ψ be an alternation-free μGF sentence, and φ a subformula of ψ. If φ is not a sentence, then let α_φ be the guard of the innermost quantifier dominating φ. Obviously $\mathrm{free}(\varphi) \subseteq \mathrm{free}(\alpha_\varphi)$. In the case where φ is a sentence, we can put $\alpha_\varphi :=$ *true*.

We now associate with every subformula φ of ψ a Datalog LITE query (Π_φ, H_φ) where H_φ has the same arity as φ such that

$$\bar{a} \in (\Pi_\varphi, H_\varphi)[\mathfrak{A}] \iff \mathfrak{A} \models \varphi(\bar{a})$$

for every structure $\mathfrak{A}$ and every tuple $\bar{a}$ that is guarded by α_φ. In the case that φ is a sentence (in particular for $\varphi = \psi$), this means that (Π_φ, H_φ) evaluates to *true* on $\mathfrak{A}$ if and only if $\mathfrak{A} \models \varphi$.

The construction is by induction on φ. As in the translation from the alternation-free portion of the μ-calculus into modal Datalog, we eliminate greatest fixed points and transform the formula so that negation signs only appear in front of LFP-operators and atoms. Since the original sentence does not have alternations between least and greatest fixed points, it follows that in subformulae of the form $[\mathrm{LFP}\ X\bar{x}\,.\,\varphi(W,\bar{x})](\bar{x})$ the fixed point variable W does not occur in the scope of any negation sign.

— If φ is an atom or a negated atom, then Π_φ consists of the single rule $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x},\bar{y}), \varphi(\bar{x})$.

— The rules for formulae of the form $\eta \vee \vartheta$, $\eta \wedge \vartheta$ and $\neg\vartheta$ are as usual, except that we always include the appropriate guard in the body of the rule (this is necessary to ensure that the rule is guarded):

—$\varphi = \eta \wedge \vartheta$: Add the rule $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x}, \bar{y}), H_\eta(\bar{x}'), H_\vartheta(\bar{x}'')$. (Here $\bar{x}'$ and $\bar{x}''$ are subtuples of $\bar{x}$.)

—$\varphi = \eta \vee \vartheta$: Add the rules $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x}, \bar{y}), H_\eta(\bar{x}')$ and $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x}, \bar{y}), H_\vartheta(\bar{x}'')$.

—$\varphi = \neg\vartheta$: Add a new stratum, consisting of the rule $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x}, \bar{y}), \neg H_\vartheta(\bar{x})$.

— For $\varphi = (\exists \bar{y}.\alpha_\vartheta)\vartheta$, add the rule $H_\varphi(\bar{x}) \leftarrow \alpha_\vartheta(\bar{x}, \bar{y}), H_\vartheta(\bar{x}', \bar{y}')$ (where again, $\bar{x}'$,$\bar{y}'$ are subtuples of $\bar{x}$,$\bar{y}$).

— For $\varphi = (\forall \bar{z}.\alpha_\vartheta)\vartheta$, add the rule $H_\varphi(\bar{x}) \leftarrow \alpha_\varphi(\bar{x}, \bar{y}), (\forall \bar{z}, \alpha_\vartheta)H_\vartheta(\bar{x}', \bar{z}')$.

— Finally, let $\varphi = [\mathrm{LFP}\ W\bar{x} \,.\, \vartheta(W, \bar{x})](\bar{x})$. By induction hypothesis, we have a Datalog LITE program Π_ϑ, that computes the update operator defined by $\vartheta(W, \bar{x})$. Since W does not appear in ϑ inside the scope of any negation sign, W is used only in the top stratum of Π_ϑ. The program Π_φ is then obtained by replacing in Π_ϑ both W and H_ϑ by H_φ.

It is straightforward to verify that (Π_φ, H_φ) indeed satisfies the required property.

For the converse, we use the fact that every Datalog LITE program is equivalent to a program where all guards are input atoms (see Theorem 4.14). From there, the translation into alternation-free μGF is completely analogous to the translation of modal programs into the μ-calculus. □

A closer look at these translation reveals that, not surprisingly, guarded first-order sentences are translated into recursion-free programs and vice versa.

THEOREM 8.6. *Recursion-free Datalog LITE has the same expressive power as the guarded fragment of first order logic.*

The translation from alternation-free μGF to Datalog LITE can be done in linear time if the underlying vocabulary is fixed or if the arity of the input relations remains bounded. This gives a simple proof that both the data complexity and expression complexity of the model checking problem of alternation-free μGF are in deterministic linear time.

For formulae with relations of unbounded arity, the translation is quadratic. Indeed, the number of rules of Π_ψ is linear in the number of subformulae of ψ, and each rule has bounded length in the case of bounded arity and linear length in the case of unbounded arity. The intuitive reason for the possibly non-linear increase of the length of the Datalog LITE program is due to the necessity to add the guards α_φ to the body of the rules. A simple example is a formula of the form $(\exists \bar{y} \,.Rx\bar{y}) \bigvee_{i=1}^{n} Sy_iy_{i+1}$ which is translated into a program consisting of n rules, each of the form $H_\psi x \leftarrow Rx\bar{y}, Sy_iy_{i+1}$.

However, also the *combined complexity* of alternation-free μGF model checking is linear in both arguments (even if the vocabulary varies with the input). This has been shown in [Berwanger 2000] by a reduction to parity games. A parity game is given by a game graph $G = (V, E, f, v_0)$ with set of positions $V = V_0 \cup V_1$, possible moves given by $E \subseteq V \times V$, a priority function $f : V \to \mathbb{N}$ on the nodes, and an initial position v_0. The game is played by two players, 0 and 1, starting from position v_0. At positions in V_0, player 0 moves, at positions in V_1, player 1 moves; if a player cannot move, she looses. The winner of infinite plays are determined according to the parity condition: player 0 wins if the minimal priority appearing infinitely often in the play is even, otherwise player 1 wins.

It is known that parity games are determined (from each position, one of the players has a winning strategy) and in fact, one can always find positional winning strategies (i.e. the decision how to move next depends only on the current position, not on the history of the play). As a consequence the strategy problem for parity games is in NP $\cap$ Co-NP. It is open whether this problem can be solved in polynomial time. The best known deterministic algorithms compute winning positions and winning strategies in time $O\Big(|E|\Big(\frac{|V|}{k/2}\Big)^{k/2}\Big)$ and space $O(k|V|)$ where k is the maximal priority in the given game [Jurdziński 2000].

Solving parity games is in some sense the computational essence of evaluating fixed point formulae, at least if nested least and greatest fixed points must be handled. In the case of guarded fixed-point formulae the reduction from model-checking to parity games is particularly pleasing, since it it can be computed in linear time: Given an arbitrary μGF-sentence ψ and a finite structure $\mathfrak{A}$, one can construct in time $O(||\mathfrak{A}|| \cdot |\psi|)$ an instance of the parity game which is won by player 0 if and only $\mathfrak{A} \models \psi$. The priorities in the game are very closed related to the the alternation depths of the fixed point formulae in ψ. In particular, for alternation-free formulae one gets parity games that can be solved in linear time (see [Berwanger 2000]).

THEOREM 8.7. *The combined complexity of the model checking problem for alternation-free* μGF *is linear both with respect to the size of the structure and the length of the formula.*

We recall that it is still open whether the model checking problem for the μ-calculus with unbounded alternations can be solved in polynomial time. Again, a simple reduction argument shows that if this should be the case, then also μGF admits polynomial time model checking.

9. CONCLUSION AND CURRENT WORK

There are several interesting technical questions related to Datalog LITE which we are currently studying. One is to find suitable extensions of this language to express CTL* and LTL. This can be done by adding very few new primitives along the lines of a general approach to extending Datalog by generalized quantifiers and subprogams [Eiter et al. 1997; 1999]. Using the same methods, it is also possible to obtain stronger variants of Datalog LITE which can express fixed point alternations, and thus, the modal μ-calculus. An efficient algorithm for evaluating alternating fixed points in a logic programming framework has been developed by Liu, Ramakrishnan and Smolka [Liu et al. 1998]. Another interesting issue currently under investigation is the relationship between Datalog LITE and automata, and more general, Datalog and tree automata.

We plan to use Datalog LITE not only for databases, but also for verification. We believe that the semantic proximity to temporal logics and the fixed point semantics of Datalog LITE make it possible to extend most of the techniques developed for temporal logics to Datalog LITE. As the scope of verification increasingly broadens, we expect that for verification of software and e-commerce protocols as in [Abiteboul et al. 1998], expressive hybrid languages such as Datalog LITE will be of great value.

ACKNOWLEDGMENTS

The authors thank the anonymous referees, Edmund Clarke, Martin Grohe, Jörg Flum, Thomas Henzinger, Sergey Vorobyov, and Harald Ganzinger for helpful comments.

REFERENCES

ABADI, M. AND MANNA, Z. 1989. Temporal logic programming. *Journal of Logic Programming 8(3)*, 277 – 295.

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

ABITEBOUL, S., VIANU, V., FORDHAM, B. S., AND YESHA, Y. 1998. Relational transducers for electronic commerce. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, J. Paredaens, Ed. ACM Press, 179–187.

ANDRÉKA, H., VAN BENTHEM, J., AND NÉMETI, I. 1998. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic 27*, 217–274.

APT, K. R., BLAIR, H. A., AND WALKER, A. 1988. Towards a theory of declarative knowledge. In *Foundations of DD and LP*, J. Minker, Ed. 89–148.

BAUDINET, M. 1989a. Logic programming semantics: Techniques and applications. PhD Thesis, Stanford University.

BAUDINET, M. 1989b. Temporal logic programming is complete and expressive. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 276–280.

BAUDINET, M. 1992. A simple proof of the completeness of temporal logic programming. In *Intensional Logics for Programming*, L. F. del Cerro and M. Penttonen, Eds. Oxford University Press.

BAUDINET, M. 1995. On the expressiveness of temporal logic programming. *Information and Computation 117(2)*, 157–180.

BAUDINET, M., CHOMICKI, J., AND WOLPER, P. 1993. Temporal deductive databases. In *Temporal Databases*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds. Benjamin/Cummings.

BERMAN, K. A., SCHLIPF, J. S., AND FRANCO, J. V. 1995. Computing well-founded semantics faster. In *LPNMR'95*, V. Marek and A. Nerode, Eds. LNCS. Springer, 113–126.

BERWANGER, D. 2000. Games and model checking for guarded logics. Diplomarbeit, RWTH Aachen.

BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *TACAS*. 193–207.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98(2)*, 142–170.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic Programming and Databases*. Surveys in Computer Science. Springer.

CHARATONIK, W. AND PODELSKI, A. 1998. Set-based analysis of reactive infinite-state systems. In *TACAS'98*, B. Steffen, Ed. LNCS, vol. 1384. Springer.

CHOMICKI, J. 1990a. Functional deductive databases: Query processing in the presence of limited function symbols. PhD Thesis, Rutgers University.

CHOMICKI, J. 1990b. Polynomial- time computable queries in temporal deductive databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 379–391.

CHOMICKI, J. AND IMIELIŃSKI, T. 1988. Temporal deductive databases and infinite objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 61–73.

CHOMICKI, J. AND IMIELIŃSKI, T. 1989. Relational specifications of infinite query answers. In *ACM SIGMOD International Conference on Management of Data*. 174–183.

CLARKE, E. AND EMERSON, E. A. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs: Workshop*. LNCS, vol. 131. Springer, 52–71.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *Computer-Aided Verification (CAV) 2000*. LNCS, vol. 1855. Springer. Full version available as Technical Report CMU-CS-00-103, Carnegie Mellon University.

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

CLARKE, E. AND SCHLINGLOFF, H. 2000. Model checking. In *Handbook of Automated Reasoning*, J. Robinson and A. Voronkov, Eds. Elsevier. to appear.

CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS) Vol.16,* 5 (September), pp.1512 – 1542.

CODD, E. 1972. Relational Completeness of Database Sublanguages. In *Courant Computer Science Symposium 6: Database Systems*, R. Rustin, Ed. Vol. 3. Prentice-Hall, Englewood Cliffs, NJ, 65–98.

CONSENS, M. P. AND MENDELZON, A. O. 1993. Low-complexity aggregation in GraphLog and Datalog. *Theor.Comp.Science 116*, 95–116.

COURCELLE, B. 1990. Graph rewriting: An algebraic and logic approach. In *Handbook of Theor.Comp.Science. Vol. B*. Elsevier, 193–242.

CUI, B., DONG, Y., DU, X., KUMAR, K. N., RAMAKRISHNAN, C. R., RAMAKRISHNAN, I. V., ROYCHOUDHURY, A., SMOLKA, S. A., AND WARREN, D. S. 1998. Logic programming and model checking. In *PLAP/ALP'98*, C. Palamidessi, H. Glaser, and K. Meinke, Eds. LNCS, vol. 1490. Springer, 1–20.

DAHLHAUS, E. 1987. Skolem normal forms concerning the least fixpoint. In *Computation Theory and Logic*. LNCS, vol. 270. Springer, 101–106.

DANTSIN, E., EITER, T., GOTTLOB, G., AND VORONKOV, A. 1997. Complexity and expressive power of logic programming. In *Proc. 12th IEEE Conference on Computational Complexity (CCC'97)*. 82–101. Extended version to appear in *ACM Computing Surveys*.

DOWLING, W. F. AND GALLIER, J. H. 1984. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Logic Programming 1,* 3 (Oct.), 267–284.

EBBINGHAUS, H.-D. AND FLUM, J. 1999. *Finite Model Theory (2nd edition)*. Springer.

EITER, T., GOTTLOB, G., AND VEITH, H. 1997. Modular logic programming and generalized quantifiers. In *LPNMR'97*, J. Dix, U. Furbach, and A. Nerode, Eds. LNCS, vol. 1265. Springer, 290–309.

EITER, T., GOTTLOB, G., AND VEITH, H. 1999. Generalized Quantifiers in Logic Programs. In *Proceedings of the ESSLLI Workshop on Generalized Quantifiers, Aix-en-Provence*, J. Väänänen, Ed. LNCS, vol. 1754. Springer.

EMERSON, E. 1990. Temporal and modal logic. In *Handbook of Theor.Comp.Science. Vol. B.*, J. van Leeuven, Ed. Elsevier, 995–1072.

FLUM, J. 1999. On the (infinite) model theory of fixed-point logics. In *Models, algebras, and proofs*, X. Caicedo and C. Montenegro, Eds. Number 2003 in Lecture Notes in Pure and Applied Mathematics Series. Marcel Dekker, 67–75.

GELDER, A. V., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38,* 3 (July), 620–650.

GOTTLOB, G., GRÄDEL, E., AND VEITH, H. 2000. Linear time datalog and branching time logic. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer. To appear.

GOTTLOB, G., LEONE, N., AND VEITH, H. 1999. Succinctness as a source of complexity in logical formalisms. *Annals of Pure and Applied Logic 97(1-3)*, 231–260.

GRÄDEL, E. 1992. On transitive closure logic. In *CSL 91*. LNCS, vol. 626. Springer-Verlag, 149–163.

GRÄDEL, E. 1999a. Decision procedures for guarded logics. In *Automated Deduction - CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*. Lecture Notes in Artificial Intelligence, vol. 1632. Springer-Verlag.

GRÄDEL, E. 1999b. On the restraining power of guards. *Journal of Symbolic Logic 64*, 1719–1742.

GRÄDEL, E. 1999c. Why are modal logics so robustly decidable? *Bulletin of the European Association for Theoretical Computer Science 68*, 90–103.

GRÄDEL, E. AND WALUKIEWICZ, I. 1999. Guarded fixed point logic. In *Proc. 14th IEEE Symp. on Logic in Computer Science*, G. Longo, Ed. 45–54.

IMMERMAN, N. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput. 17(5)*, 935–938.

IMMERMAN, N. AND VARDI, M. Y. 1997. Model checking and transitive-closure logic. In *CAV 1997*, O. Grumberg, Ed. LNCS, vol. 1254. Springer, 291–302.

ITAI, A. AND MAKOWSKY, J. A. 1987. Unification as a complexity measure for logic programming. *J. of Logic Programming 4,* 2 (June), 105–117.

JANIN, D. AND WALUKIEWICZ, I. 1996. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR 96*. LNCS, vol. 1119. Springer-Verlag, 263–277.

JURDZIŃSKI, M. 2000. Small progress measures for solving parity games. In *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*. Lecture Notes in Computer Science, vol. 1770. Springer-Verlag, 290–301.

KOLAITIS, P. G. 1990. Implicit definability on finite structures and unamboguous computations. *Inf.Comp. 90*, 50–66.

KOZEN, D. 1983. Results on the propositional μ-calculus. *Theor.Comp.Science 27,* 3 (Dec.), 333–354.

KURSHAN, R. P. 1994. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press.

LIU, X., RAMAKRISHNAN, C., AND SMOLKA, S. 1998. Fully local and efficient evaluation of alternating fixed points. In *TACAS'98*. LNCS, vol. 1384. Springer.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer.

MINOUX, M. 1988. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Inf.Proc.Let. 29,* 1 (15 Sept.), 1–12.

MORET, B. AND SHAPIRO, H. 1990. *Algorithms from P to NP*. Benjamin/Cummings.

MURAKAMI, M. 1990. A declarative semantics of flat guarded Horn clauses for programs with perpetual processes. *Theor.Comp.Science 75,* 1–2 (25 Sept.), 67–83.

PELED, D. A., PRATT, V. R., AND HOLZMANN, G. J., Eds. 1997. *Partial Order Methods in Verification*. DIMACS series, vol. 29. American Mathematical Society.

RAMAKRISHNAN, Y. S., RAMAKRISHNAN, C. R., RAMAKRISHNAN, I. V., SMOLKA, S. A., SWIFT, T., AND WARREN, D. S. 1997. Efficient model checking using tabled resolution. In *CAV'97*, O. Grumberg, Ed. LNCS, vol. 1254. Springer, 143–154.

SEESE, D. 1996. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science 6,* 6 (Dec.), 505–526.

ULLMAN, J. D. 1989. *Principles of Data Base Systems*. Computer Science Press.

VAN EMDE BOAS, P. 1990a. Machine models and simulation. In *Handbook of Theoretical Computer Science*, J. van Leeuven, Ed. Vol. A. Elsevier, 3–66.

VAN EMDE BOAS, P. 1990b. Machine models and simulations. In *Handbook of Theor.Comp.Science. Vol. A.*, J. van Leeuven, Ed. Elsevier, 1–66.

VARDI, M. 1982. Complexity of Relational Query Languages. In *Proceedings 14th STOC*. San Francisco, 137–146.

VARDI, M. AND WOLPER, P. 1994. Reasoning about infinite computations. *Information and Computation 115(1)*, 1–37.

VARDI, M. Y. 1998. Reasoning about the past with two-way automata. In *ICALP*, K. G. Larsen, S. Skyum, and G. Winskel, Eds. LNCS, vol. 1443. Springer, 628–641.

WOLPER, P. 1983. Temporal logic can be more expressive. *Information and Control 56,* 1–2, 72–99.

Received June 2000; revised November 2000; accepted January 2001