Chapter 1

LINEAR TIME DATALOG AND BRANCHING TIME LOGIC

Georg Gottlob
Institut für Informationssysteme
Technische Universität Wien, Austria
gottlob@dbai.tuwien.ac.at

Erich Grädel
Mathematische Grundlagen der Informatik
RWTH Aachen, Germany
graedel@informatik.rwth-aachen.de

Helmut Veith
School of Computer Science
Carnegie Mellon University, Pittsburgh, USA
veith@cs.cmu.edu

Abstract We survey recent results about the relation between Datalog and temporal verification logics. Datalog is a well-known database query language relying on the logic programming paradigm. We introduce *Datalog LITE*, a fragment of Datalog with well-founded negation, which has an easy stratified semantics and a linear time model checking algorithm. Datalog LITE subsumes temporal languages such as CTL and the alternation-free μ-calculus. We give easy syntactic characterizations of these temporal languages by fragments of Datalog LITE, and show that Datalog LITE has the same expressive power as the alternation-free portion of guarded fixed point logic.

Keywords: Datalog, temporal logic, computer-aided verification, linear time

1. INTRODUCTION

During the last decade, model checking has become one of the preeminent applications of logic in computer science. Temporal model checking is a technique for verifying that a system satisfies its specifications 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 has become an industrial standard tool for hardware design.

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 number of variables. 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), abstraction techniques (e.g. Clarke et al., 1994; Kurshan, 1994; Clarke et al., 2000a), and reduction techniques such as the cone of influence reduction and the partial order reduction. For recent overviews of model checking, we refer the reader to Clarke et al., 2000b; Clarke and Schlingloff, 2000.

The temporal logics used in model checking provide a number of important properties which are crucial for the method, distinguish them from many other logics in computer science, and serve as a basis for the above mentioned methods. The most outstanding properties are fast (in fact, often linear time) model checking procedures, decidability of the satisfiability problem, and structure theorems about the models of temporal formulas such as the tree model property. Some of the most prominent examples of temporal logics are branching time logics such as computational tree logic CTL (Clarke and Emerson, 1981), the modal μ-calculus L_μ (Kozen, 1983), and its alternation-free fragment L_{μ_1}.

The contribution of this paper is to present Datalog LITE (**LI**near **T**ime **E**xtended Datalog), a powerful language based on logic programming which subsumes CTL and L_{μ_1}. Datalog LITE is a deductive query language which 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. Thus, Datalog LITE has potential applications in verification systems as well as in databases. It is expected that for verification of software and e-commerce protocols as in Abiteboul et al., 1998, such hybrid languages may be of great value.

Example 1 *In the verification system SMV, the state space S of a Kripke structure is given by the possible assignments to the system variables. Thus, a system with 3 variables $x, y, reset$ and variable domains $D_x = D_y = \{0,1,2,3\}$ and $D_{reset} = \{0,1\}$ has state space $S = D_x \times D_y \times D_{reset}$, and $|S| = 32$.*

A binary transition relation E is defined by transition blocks which for each variable define its possible next value in the next time cycle, as in the following example:

```
init(reset) := 0;
next(reset) := {0,1};
```

```
init(x) := 0;
next(x) := case
    reset = 1 : 0;
    x < y : x + 1;
    x = y : 0;
    else : x;
esac;
```

```
init(y) := 1;
next(y) := case
    reset = 1 : 0;
    (x = y) ∧ ¬(y = 2) : y + 1;
    (x = y) : 0;
    else : y;
esac;
```

Here, **next**$(reset) := \{0,1\}$ *means that the value of reset is chosen nondeterministically. Such situations occur frequently when reset is determined by the environment, or when the model of the system is too abstract to determine the values of reset. For details about the SMV input language, we refer the reader to McMillan, 1993. Typical CTL properties to be verified by the system include the following:*

CTL	*Informal Semantics*
AGEF $reset = 1$	*"From all reachable states, it is possible to reset the system in the future."*
EFAG $x = 1$	*"There exists a reachable state, after which $x = 1$ becomes an invariant."*

The formal semantics of CTL is defined in Section 2.

In database theory, a database is identified with a finite relational structure. 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. A Datalog query (Π, P) is a finite set of (recursive or nonrecursive) rules Π (i.e., a Datalog program) and a distinguished output relation P. Since queries are defined by Datalog programs, we shall usually not distinguish between queries and programs.

Example 2 *The query* (Π, T)*, where* Π *consists of the rules*

$$T(x,y) \leftarrow E(x,y) \qquad\qquad T(x,y) \leftarrow T(x,z), E(z,y).$$

associates with any input relation E *its transitive closure.*

A brief description of Datalog is given in Section 3. 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* introduced by Apt et al., 1988 where negation does not interact with recursion, and the more expressive Datalog with *well-founded negation* defined by Gelder et al., 1991.

Example 3 *We continue Example 1. In order to evaluate a Datalog program over a Kripke structure, we use the transition relation* E *as input relation. Moreover, for all atomic properties of states* s *such as* $x = 1$*, monadic relations such as* $I_{x=1}$ *are provided with the input. For better readability, we shall write* $s.x = 1$ *instead of* $I_{x=1}(s)$*.*

Then the specifications of Example 1 can be expressed in Datalog as follows:

AGEF $reset = 1$	$\text{Reset\_Reachable}(s) \leftarrow s.reset = 1$
	$\text{Reset\_Reachable}(s) \leftarrow E(s,s'), \text{Reset\_}Reachable(s')$
	$\text{BadState}(s) \leftarrow \neg\text{Reset\_Reachable}(s)$
	$\text{BadState}(s) \leftarrow E(s,s'), \text{BadState}(s')$
	$\text{GoodState}(s) \leftarrow \neg\text{BadState}(s)$
EFAG $x = 1$	$\text{No\_Invariant}(s) \leftarrow \neg s.x = 1$
	$\text{No\_Invariant}(s) \leftarrow E(s,s'), \text{No\_Invariant}(s')$
	$\text{Result}(s) \leftarrow \neg\text{No\_Invariant}(s)$
	$\text{Result}(s) \leftarrow E(s,s'), \text{Result}(s')$

The above programs belong to a simple fragment of both Datalog LITE and Stratified Datalog. While at first sight these programs are longer than their counterparts in CTL, they have several advantages including easy readability and reusability of (sub)programs. Moreover, we shall see that the expressive power of Datalog LITE is much higher than that of CTL and L_{μ_1}*.*

Research Programme. The favorable logical properties of temporal logics have attracted interest from various communities. In pure logic, modal fragments of first order logic and generalizations thereof such as

the bounded fragment have been studied by Andréka et al., 1998; Grädel, 2000; Grädel and Walukiewicz, 1999, and general insight in model theoretic and algorithmic properties of temporal logics has been gained.

For databases, fast model checking procedures and decidability are natural properties of query languages which have received wide consideration in the literature. The advantages of Datalog and its capability of expressing and checking temporal properties of Kripke structures suggest to study fragments of Datalog that are suited for model checking. We believe that from the point of view of both verification and databases it is a rewarding task to investigate the exact relation between temporal logics and deductive query languages. Thus, our research programme was aimed at identifying a suitable Datalog fragment which meets the following criteria:

- The fragment should be clearly syntactically identifiable, but syntactically ample enough to allow the programmer to make use of the main paradigms of Logic Programming (including, in particular, some form of negation).
- It should be expressive enough to state a large number of useful temporal properties, in particular those in CTL.
- The fragment should have linear time data complexity. This means that the complexity of evaluating a fixed query over variable input structures is $O(n)$, where n is the structure size.
- The fragment 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.
- Given the increasing importance of model checking on infinite transition systems, the fragment should be able to express relevant temporal properties not just on finite Kripke structures but also on infinite ones.

In the current paper we survey recent results in this direction. We refer the interested reader to the technical report (Gottlob et al., 1998) which contains all proofs and more technical details.

Note that a mere *ad hoc* translation of CTL to Datalog does not provide us with a suitable fragment fulfilling the above requirements. The fragment suggested by such a translation is – on one hand – syntactically extremely limited, e.g., no predicate symbols of arity greater than two occur in it. On the other hand this fragment is not known to admit a linear time evaluation algorithm.

Results. We present *Datalog LITE*, a fragment of Datalog that fulfills all desired criteria. 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.

We prove that Datalog LIT has linear time data complexity (Theorem 8). 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 3., Theorem 10), 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 $(\forall y_1 \cdots y_n.\alpha)\beta$ is its standard first order semantics, that is, $\forall y_1 \cdots y_n(\alpha \rightarrow \beta)$. (For a precise definition, see Section 3.) It is important to note that generalized literals are not just artificial "plug ins" to Datalog, but that 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. Datalog LITE has the following important properties:

- For fixed Datalog LITE programs model-checking is feasible in linear time (Theorem 18). This remains true even for variable Datalog LITE programs whose predicate-arities are bounded by a constant or whose guards are all input atoms.

- Datalog LITE can be easily embedded into well-founded Datalog. Thus, Datalog LITE can be considered an intuitive yet powerful fragment of well-founded Datalog with linear time model checking.

- We show that semantically, every Datalog LITE program is equivalent to a Datalog LITE program where all guards are input

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

atoms; for arbitrary programs, the translation requires exponential time. In fact, evaluating variable Datalog LITE programs with unbounded predicate arities is EXPTIME- complete. (Proposition 19).

- 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, 1999; several important properties like satisfiability in EXPTIME and a generalized tree model property are thus obtained for Datalog LITE.

- Propositional multi-modal logic, CTL, the alternation free modal μ-calculus, and guarded first order logic each correspond to well-defined and syntactically simple subfragments of Datalog LITE.

Hence, Datalog LITE (with bounded arities) is at least as expressive as CTL and the alternation-free μ-calculus L_{μ_1}. Actually Datalog LITE is more expressive than these logics. For example, it is obvious that *past modalities* (cf. Vardi, 1998) are expressible in Datalog LITE. Furthermore Datalog LITE can define new modalities, e.g. via Boolean combinations of existing ones.

Related work. The definitions of guarded rules and generalized literals are motivated by the notion of *guarded quantification* as introduced by Andréka et al., 1998 (see Section 7.) 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, cf. Grädel and Walukiewicz, 1999.

It is not new that CTL, the modal μ-calculus and related formalisms can be translated into logic programming. Such translations were carried-out by Ramakrishnan et al., 1997; Cui et al., 1998; Charatonik and Podelski, 1998. In a recent paper by Immerman and Vardi, 1997 it is shown how various temporal logics can be translated into first order logic augmented by a transitive closure construct (FO+TC). Since the latter is expressible in Datalog, that approach implicitly yields a translation into Datalog. 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.

2. 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; Clarke and Schlingloff, 2000. The formulae of the logics we describe here are evaluated on Kripke structures (with one or more transition relations) at a particular state.

A Kripke structure $\mathfrak{K}$ is usually given as a tuple $(S, (E_a)_{a\in A}, L)$ where S is the state space, A is a set of actions, E_a is the transition relation associated with action a, and $L : S \to \{P_1, \ldots, P_k\}$ is a labelling function that assigns a set of atomic propositions to each state. $\mathfrak{K}$ can be trivially identified with the relational structure $(S, (E_a)_{a\in A}, P_1, \ldots P_k)$ where $P_1, \ldots, P_k$ are monadic relations representing all the atomic propositions and for each $s \in S$, $P_i(s)$ is true iff $P_i \in L(s)$.

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. 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.

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, hence one transition relation in the Kripke structure one simply writes $\Box$ and $\Diamond$ for the modal operators.

Semantics of ML. Let ψ be a formula of ML, $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, P_2, \ldots)$ 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. (Note that this definition of totality which is common in model checking is differs from standard terminology.)

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 only 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 $\mathcal{K} = (S, E, P_1, P_2, \ldots)$. An *infinite E-path* (or just an infinite path) is an infinite sequence $\bar{s} = s_0, s_1, s_2, \ldots$ 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 $s_i, s_{i+1}, s_{i+2}, \ldots$. 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.

(S3) $\mathcal{K}, s_0 \models \mathbf{E}\vartheta$ 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:

- $\mathbf{EF}\psi$ abbreviates $\mathbf{E}(\text{true}\mathbf{U}\psi)$,
- $\mathbf{AG}\psi$ abbreviates $\neg\mathbf{EF}\neg\psi$,
- $\mathbf{AF}\psi$ abbreviates $\mathbf{A}(\text{true}\mathbf{U}\psi)$, and
- $\mathbf{EG}\psi$ abbreviates $\neg\mathbf{AF}\neg\psi$.

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 $X, Y, \ldots$) 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.

An L_μ-formula is *alternation-free* if no occurrence of any propositional variable is inside the scope of both a μ- and a ν-operator.

***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 $\psi^{\mathcal{K}}$.

3. FROM DATALOG TO DATALOG LIT

Definition 4 A *Datalog rule* is an expression of the form $H \leftarrow B_1, \ldots, B_r$, $r \geq 1$, where H, the *head* of the rule, is an atomic formula $Ru_1 \cdots u_s$, and $B_1, \ldots, B_r$, 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 $u_1, \ldots, u_s, v_1, \ldots, v_t$ are variables. The relation symbol R is called the *head predicate* of the rule. Note that we use the symbol $\neg$ to denote default negation as common in database theory; in AI, the symbol *not* is more common.

We first define *basic Datalog programs* where negation is applied only to input relations. 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 structure $\mathfrak{A}$ over the vocabulary of the extensional 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, we refer to Abiteboul et al., 1995.

Note that atoms may be written in the form Axy or in the form $A(x, y)$. Usually, we shall prefer the first form in the context of proofs and definitions (where atoms are usually denoted by single letters), and the second form in real programs as in Example 3, where atoms are denoted by longer names like Invariant.

Definition 5 A *stratified Datalog program* is a sequence $\Pi = (\Pi_0, \ldots, \Pi_r)$ of basic Datalog programs which are called the *strata* of Π, such that Π_0 is a basic Datalog program, and every Π_i is a basic Datalog program whose extensional predicates are either extensional in the entire program Π or are head predicates of a lower stratum. Thus, 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 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 by 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.

Definition 6 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.

We adopt the convention that the input relations of a query are given by lists of tuples. We shall refer to this representation as the *list representation* of the input structure.

Definition 7 Let $\mathfrak{A}$ be a finite 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 algorithms 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, 1990), but on the other hand, we obtain a non-realistic notion of linear time computability when the adjacency matrix is sparse. It is easy to see that a Random Access Machine can translate the list representation into array representation in linear time.

Theorem 8 *The time complexity of model checking (i.e., evaluation) of Datalog LIT is*

1. *linear in the input size and the program size for programs where all guards are input relations.*

2. *linear in the input size and the program size for bounded arity programs.*

3. *linear in the input size and exponential in the program size for arbitrary programs.*

We defer the proof to Section 5., where it follows from a more general result. In the rest of this section we show that Datalog LIT is too restricted to express important linear time computable properties. We exemplify these restrictions by the well-foundedness query which is linear time computable and important in many applications. (Recall that a

vertex x is well-founded, if no cycle can be reached from x, and no infinite path originates at x.)

Example 9 *Well-foundedness statements for subgraphs of the transition relation 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).*

Well-foundedness in finite graphs. It is easy to see that in finite graphs, detecting well-foundedness essentially reduces to detecting cycles in the graph; in full stratified Datalog, this is done by defining the transitive closure of the graph, i.e. by using a binary relation as intermediate result. The following results state that this is not possible in Datalog LIT, and that indeed Datalog LIT does not express well-foundedness.

Theorem 10 *1. Over finite structures, the well-foundedness query cannot be expressed by stratified Datalog programs with only monadic head predicates.*

2. Over Kripke structures, every Datalog LIT query with monadic output predicate is equivalent to a Datalog LIT query where all head predicates are monadic.

Proof of 1. 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 let ρ be 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 colours, 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 coloured identically, and the j-cycle in $\mathfrak{B}$ is coloured 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 colours of that stratum. By induction hypothesis the m-neighbourhoods around corresponding elements of $\mathfrak{A}$ and $\mathfrak{B}$ satisfy the same existential statements concerning the old colours. Therefore, for sufficiently large i, j, k, the loop and the copies of S_j are coloured isomorphically. Operationally, one can imagine that first the substructure $\mathfrak{A}$ of $\mathfrak{B}$ is coloured, and then the loop has to be coloured identically, since the existential statements are true on the loop if they were true on S_j. $\square$

Corollary 11 *Datalog LIT does not express the well-foundedness query. Hence, Datalog LIT does not express all of CTL.*

Well-foundedness in infinite graphs. The intuitive reason why, over infinite structures, well-foundedness is 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 12 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 13 *Every query in stratified Datalog is equivalent to some formula in* $\bigcup_{k<\omega} L(\Sigma_k)$.

Theorem 14 *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.*

4. 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 15 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 $\text{free}(\alpha) \supseteq \text{free}(\beta)$. The free variables $\text{free}(G)$ of G are given by $\text{free}(\alpha) - \{y_1, \ldots, y_n\}$.

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 to be regarded as 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. In the context of a rule, the generalized literal is regarded as a negative literal. Since the notion of free variables is well-defined for generalized literals, Definition 6 is applicable to rules with generalized literals.

Definition 16 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.

Note that according to this definition, generalized literals cannot be used as guards. 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

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

Remarks.

1. Datalog LITE is indeed stronger than Datalog LIT, since the program $Wx \leftarrow (\forall y.Exy)Wy$ computes the well-foundedness query on both finite and infinite structures.

2. 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 by Gelder et al., 1991 assigns the correct meaning to the query:

Theorem 17 *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; by the following result, we have thus identified a linear time computable fragment of well-founded Datalog:

Theorem 18 *The complexity results about Datalog LIT (Theorem 8) hold also for Datalog LITE.*

We shall see later (cf. Lemma 21) that surprisingly, the use of head predicates as guards does not increase the expressive power of Datalog LITE. Therefore, from the point of expressive power one can always use Datalog LITE with input guards. Head predicates as guards permit writing programs in a much more compact way. By the following proposition, the exponential time bound for this case is indeed optimal:

Proposition 19 *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 and for programs with input guards only.*

Proof Sketch. Since Datalog LITE is obviously contained in fixed point logic whose data complexity is PTIME, membership follows for both formalisms. For hardness, it is easy to see that the problem *Monotone Circuit Value* is expressible in Datalog LIT. 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 8. For general program

complexity, it is again sufficient to show hardness for Datalog LIT; since we know that Monotone Circuit Value is expressible in Datalog LIT, the general method for program complexity worked out in Gottlob et al., 1999 is applicable. For details, consult Gottlob et al., 1999, Section 6.4 and Gottlob et al., 1998. □

5. LINEAR TIME ALGORITHMS

This section outlines the proof of Theorems 8 and 18. We proceed in two phases: (1) we show that we can reduce the problem to programs with input guards only, and (2) we solve the problem for this restricted case.

Phase 1: Locality Invariance and Input Guards. Given a structure $\mathfrak{A}$, we define the binary relation $E^{\mathfrak{A}}$ on $\mathfrak{A}$ by $E^{\mathfrak{A}} = \{(a,b) :$ there is a tuple in a relation in $\mathfrak{A}$ containing both a and $b\}$. The graph $(|\mathfrak{A}|, E^{\mathfrak{A}})$ is called the *Gaifman graph* of $\mathfrak{A}$.

Lemma 20 *Let Π be a Datalog LITE program with head predicates $T_1, \ldots, T_r$, and let $T_1^{\mathfrak{A}}, \ldots, T_r^{\mathfrak{A}}$ be the relations computed by Π on input structure $\mathfrak{A}$. Then the Gaifman graphs of $\mathfrak{A}$ and $(\mathfrak{A}, T_1^{\mathfrak{A}}, \ldots, T_r^{\mathfrak{A}})$ coincide.*

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. The most important application of locality invariance is the following lemma:

Lemma 21 *Every Datalog LIT and Datalog LITE program is equivalent to a program where all guards are input atoms. Moreover, the time complexity of eliminating non-input guards is*

1. *linear in the program size if the program has bounded arity.*
2. *exponential in the program size if the program has unbounded arity.*

Proof Sketch. The proof is based on the observation that due to locality invariance, all head predicates contain only tuples which are obtained as permutations of subtuples of input relations. Hence, every rule can be replaced by new rules obtained by adding an input relation to the body as new guard, and unifying the old guard and the new guard in such a way that all the variables which occur in the old guard appear in the new guard. The number of new rules is easily seen to be exponential in the number of the old rules. The algorithm is immediately obtained from the exact proof. It is easily checked that the algorithm is

exponential in the program arity, as exemplified in the following remark. □

Example 22 *Consider the rule $Rx \leftarrow Gxyz$ where G is a guard, but not an input guard, and there is only one input relation E of arity 2. Then the rule can be replaced by the following set of new rules which are all guarded by E.*

$$\begin{array}{lcl@{\qquad}lcl} Rx & \leftarrow & Exy, Gxyy & Rx & \leftarrow & Eyx, Gxyy \\ Rx & \leftarrow & Exy, Gxxy & Rx & \leftarrow & Eyx, Gxxy \\ Rx & \leftarrow & Exy, Gxyx & Rx & \leftarrow & Eyx, Gxyx \\ Rx & \leftarrow & Exy, Gxxx & Rx & \leftarrow & Eyx, Gxxx \end{array}$$

Phase 2: Evaluation of Input Guard Programs. 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 (cf. Dowling and Gallier, 1984; Itai and Makowsky, 1987; Minoux, 1988) that propositional Horn clause programs can be evaluated by a variant of unit resolution 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.

The 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 from rule bodies, 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 as in Berman et al., 1995.

Algorithm EvalLIT. 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 $Fx \leftarrow Gx, My$, it will be replaced by $Fx \leftarrow Gx, M'$, and $M' \leftarrow My$. 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 $|\mathfrak{A}|$.

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

(2) The Datalog program is evaluated stratum by stratum as a propositional Horn program similarly as sketched by 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. Thus, the algorithm is linear both in the program size and $|\mathfrak{A}|$.

Algorithm EvalLITE. For Datalog LITE, we shall adapt the algorithm to handle generalized literals. Over an 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.

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 8), 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. (Note that the standard RAM model of van Emde Boas, 1990 assumes an array to be initialized by zeroes. Alternatively, the initialization can be simulated by the *lazy array* technique as described by Moret and Shapiro, 1990 with time overhead linear in $|\mathfrak{A}|$.)

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.

6. MODAL DATALOG AND CTLOG

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 23 A *modal* Datalog program Π is a Datalog LITE program such that

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

$$Hx \leftarrow (\neg)P_1x, \ldots, (\neg)P_rx$$
$$Hx \leftarrow Exy, (\neg)P_1y, \ldots, (\neg)P_ry$$
$$Hx \leftarrow (\forall y.Exy)P_0y, (\neg)P_1y, \ldots, (\neg)P_ry$$

Here, $(\neg)$ denotes that negation is optional.

The notion of bisimulation plays a central role in modal logic and model checking. Bisimilarity is usually defined as a kind of two-player game, cf. Clarke et al., 2000b. For our purposes, it is sufficient to know that two finite structures are bisimilar iff they cannot be distinguished by ML formulas. In fact, it is a characteristic feature of modal logics that they cannot distinguish between bisimilar structures. Thus, the following lemma justifies the terminology *modal* Datalog program.

Lemma 24 *Every query defined by a modal Datalog program is invariant under bisimulation.*

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 $\mathfrak{K}$ and all states v of $\mathfrak{K}$, $\mathfrak{K}, v \models \psi$ iff $v \in (\Pi, H)[\mathfrak{K}]$. 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.)

Definition 25 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 $Hx \leftarrow Px, Exy, Hy$ or $Hx \leftarrow Px, (\forall y\,.\,Exy)Hy$.

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

Theorem 26 *The following equivalences are valid:*

1. *non-recursive modal Datalog = propositional multi-modal logic*

2. *CTLog = CTL on total Kripke structures.*

3. *modal Datalog = alternation-free μ-calculus*

Proof of 1. and 2. (1) Given a non-recursive modal program Π with binary predicates E_a ($a \in A$) we construct for each 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 all these rules have the form $Hx \leftarrow (\neg)Px, (\neg)Qx$ or $Hx \leftarrow E_a xy, Py$, or $Hx \leftarrow (\forall y.E_a xy)Py$ where each $S \in \{P, Q\}$ is either an extensional predicate of Π (in which case $\psi_S := S$) or a head predicate for which the formula ψ_S has already been constructed. For every rule r of the first form let $\varphi_r := (\neg)\psi_P \wedge (\neg)\psi_Q$, for every rule r of the second form, let $\varphi_r := \langle a \rangle \psi_P$, and for every rule of the third form, 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 $H_\psi x = (\forall y \,.\, E_a xy) H_\varphi y$.

(2a) *Every* CTL*-sentence ψ is equivalent to a query (Π_ψ, H_ψ) where Π_ψ is a CTLog-program.*

The translation is precisely the same as above 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:

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

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 $\mathbf{E}(\neg\varphi \mathbf{U} (\neg\varphi \wedge \neg\vartheta))$ is true (see case (5)). The second stratum takes the conjunction of R with the complement of S.

(2b) *Every query computed by a CTLog-program is expressible by a* CTL*-formula.*

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 part (1) of the proof 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, **EX** and **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). $\Box$

Proposition 27 *With respect to data complexity, CTLog is* NLOGSPACE*-complete. With respect to program complexity, CTLog is* PTIME*-complete. With respect to data complexity and program complexity, modal Datalog is* PTIME*-complete.*

7. 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, 1999. The idea to consider *guarded fragments* of first-order logic and its extensions is due to 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 generalize the modal fragment of first-order logic.

Definition 28 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 $\mathrm{free}(\psi) \subseteq \mathrm{free}(\alpha) = \{\bar{x}, \bar{y}\}$, then the formulae $\exists \bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$ and $\forall \bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x}, \bar{y}))$ 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, 2000): (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, 2000) 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 29 The guarded fixed point logic μGF is obtained by adding to GF the following rules for constructing fixed-point formulae: Let ψ be a formula, W a k-ary relation variable that occurs only positively in ψ, and let $\bar{x} = x_1, \ldots, x_k$ be a k-tuple of distinct variables. Then we can build the formulae $[\mathrm{LFP}\ W\bar{x} \,.\, \psi](\bar{x})$ and $[\mathrm{GFP}\ W\bar{x} \,.\, \psi](\bar{x})$. 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.

The semantics of the fixed point formulae is the usual one, cf. Ebbinghaus and Flum, 1999. Note that ψ may contain besides $\bar{x}$ and W other free first-order and second-order variables. The fixed-point operator binds W and $\bar{x}$ but leaves all other variables free. 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. The complexity of μGF could be identified precisely (Grädel and Walukiewicz, 1999).

Theorem 30 (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.*

Theorem 31 *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*.*

Theorem 32 *Recursion-free Datalog LITE has the same expressive power as the guarded fragment of first order logic.*

8. CONCLUSION

In summary, Datalog LITE is a robust variant of Datalog with guaranteed linear-time model checking complexity and rich expressive power. This new language facilitates the direct specification of temporal properties of finite (and infinite) state systems by using the paradigm of logic programming. Since Datalog (both with stratified or well-founded negation) is a well-studied database query language for which a number of optimization methods for secondary-storage access have been developed, our results can also be used for applications, where a finite state system is stored in a relational database.

There are several interesting questions related to Datalog LITE which we are currently studying. One is to find suitable extensions of the 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 presented in Eiter et al., 1997. Another interesting issue currently under investigation is the relationship between Datalog LITE and automata, and more general, Datalog and tree automata.

Acknowledgments

This work was supported by the Austrian Science Fund Project N Z29-INF, by Deutsche Forschungsgemeinschaft (DFG), and the Max Kade Foundation. Most of this research has been carried out while the third author was with TU Wien.

We are thankful to Harald Ganzinger, Martin Grohe, Jörg Flum, Thomas Henzinger, Sergey Vorobyov, Jack Minker and an anonymous referee for valuable comments.

References

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 Paredaens, J., editor, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART*

Symposium on Principles of Database Systems, pages 179–187. ACM Press.

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 Minker, J., editor, *Foundations of DD and LP*, pages 89–148.

Berman, K. A., Schlipf, J. S., and Franco, J. V. (1995). Computing well-founded semantics faster. In Marek, V. and Nerode, A., editors, *LPNMR'95*, LNCS, pages 113–126. Springer.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 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 Steffen, B., editor, *TACAS'98*, volume 1384 of *LNCS*. Springer.

Clarke, E. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71. Springer.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000a). Counterexample-guided abstraction refinement. Technical Report CMU-CS-00-103, Computer Science, Carnegie Mellon University. Extended abstract to appear in CAV 20000.

Clarke, E., Grumberg, O., and Peled, D. (2000b). *Model Checking*. MIT Press.

Clarke, E. and Schlingloff, H. (2000). Model checking. In Robinson, J. and Voronkov, A., editors, *Handbook of Automated Reasoning*. 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):pp.1512 – 1542.

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 Palamidessi, C., Glaser, H., and Meinke, K., editors, *PLAP/ALP'98*, volume 1490 of *LNCS*, pages 1–20. Springer.

Dowling, W. F. and Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Logic Programming*, 1(3):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 Dix, J., Furbach, U., and Nerode, A., editors, *LPNMR'97*, volume 1265 of *LNCS*, pages 290–309. Springer.

Emerson, E. (1990). Temporal and modal logic. In van Leeuven, J., editor, *Handbook of Theor.Comp.Science. Vol. B.*, pages 995–1072. Elsevier.

Gelder, A. V., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650.

Gottlob, G., Grädel, E., and Veith, H. (1998). Datalog LITE: Temporal versus deductive reasoning in verification. Technical Report DBAI-TR-98-22, Institut für Informationssysteme, TU Wien.

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. (2000). On the restraining power of guards. *J. Symb.Logic*. To appear.

Grädel, E. and Walukiewicz, I. (1999). Guarded fixed point logic. In Longo, G., editor, *Proc. 14th IEEE Symp. on Logic in Computer Science*, pages 45–54.

Immerman, N. and Vardi, M. Y. (1997). Model checking and transitive-closure logic. In Grumberg, O., editor, *CAV 1997*, volume 1254 of *LNCS*, pages 291–302. Springer.

Itai, A. and Makowsky, J. A. (1987). Unification as a complexity measure for logic programming. *J. of Logic Programming*, 4(2):105–117.

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

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

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):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):67–83.

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 Grumberg, O., editor, *CAV'97*, volume 1254 of *LNCS*, pages 143–154. Springer.

Ullman, J. D. (1989). *Principles of Data Base Systems*. Computer Science Press.

van Emde Boas, P. (1990). Machine models and simulations. In van Leeuven, J., editor, *Handbook of Theor.Comp.Science. Vol. A.*, pages 1–66. Elsevier.

Vardi, M. Y. (1998). Reasoning about the past with two-way automata. In Larsen, K. G., Skyum, S., and Winskel, G., editors, *ICALP*, volume 1443 of *LNCS*, pages 628–641. Springer.