Linear Time Datalog for Branching Time Logic *

Georg Gottlob* Erich Grädel+ Helmut Veith*,†

*Institut für Informationssysteme, Technische Universität Wien, A-1040 Wien, {gottlob,veith}@dbai.tuwien.ac.at

+ Mathematische Grundlagen der Informatik, RWTH Aachen, D-52056 Aachen, graedel@informatik.rwth-aachen.de

† School of Computer Science, Carnegie Mellon University, Pittsburgh,veith@cs.cmu.edu

Abstract

In this paper we show that Datalog is well-suited as a temporal verification language. 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, and present a linear time model checking algorithm for Datalog LITE. We show that Datalog LITE subsumes temporal languages such as CTL and the alternation-free μ-calculus, and in fact give easy syntactic characterizations of these temporal languages. We prove that Datalog LITE has the same expressive power as the alternation-free portion of guarded fixed point logic.

1 Introduction

Model Checking. Temporal Model Checking [Clarke Jr. et al., 1986] is a technique for verifying that a system satisfies its specification by (1) representing the system as a Kripke structure, (2) writing its specification in a suitable temporal logic, and (3) checking that the Kripke structure is a model of the specification formula. The most prominent examples of such temporal logics are the computational tree logic CTL, the modal μ-calculus L_μ [Kozen, 1983], its alternation-free fragment L_{μ_1}, and the linear temporal logic LTL. All these logics are extensions of modal propositional logic ML (with one or more transition relations). The success of the above-mentioned formalisms is mainly due to the following reasons: (i) a large number of *relevant* systems properties can be expressed in these logics; and (ii) these properties can be checked very efficiently, and in fact, often in linear time [Clarke Jr. et al., 1986, Cleaveland, 1993].

Datalog. 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. [Ullman, 1989, Ceri et al., 1990] 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 and a distinguished output relation P.

*Due to space restrictions, this overview omits most proofs. For details, the technical report [Gottlob et al., 1998] is available from the authors. Partially 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.

Example 1 *The transitive closure of a directed graph* $G = (V, E)$ *can be computed by the query* (Π, T)*, where* Π *consists of the rules*

$$Txy \leftarrow Exy \qquad\qquad Txy \leftarrow Txz, Ezy.$$

A brief description of Datalog is given in Section 2. 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]. The advantages of the Datalog query languages are:

- Datalog is a purely declarative query language with a clear and simple semantics; it facilitates *modular* specifications, where intermediate properties can be expressed by separate groups of rules and reused.
- Datalog has polynomial data complexity, i.e., fixed Datalog programs can be evaluated in polynomial time over input databases. This also holds for Datalog with stratified and with well-founded negation.
- Datalog is a query language suited for main-memory databases as well as for databases stored on disk. Several Datalog query processors interacting directly with secondary storage have been developed. Sophisticated query-optimization strategies (e.g. the magic set method [Bancilhon et al., 1986]) for minimizing disk access have been developed and implemented, cf. [Ceri et al., 1990].

Expressing temporal properties in Datalog. A Kripke structure $\mathfrak{K} = (S, (E_a)_{a \in A}, L)$ where S is the state space, A is a set of actions, E_a is the state transition relation associated with action A, and L is a labeling function that assigns a set of atomic propositions to each state, 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 $P_i \in L(s)$. Thus, Datalog queries can be issued over Kripke structures. The following example shows that interesting temporal properties can be expressed this way.

Example 2 *Consider a query* (Π, H) *over a relational Kripke structure* (S, E, P, Q) *expressed in Datalog with well-founded negation, 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* $\mathbf{E}(\neg P \mathbf{U} Q)$.

Research Programme. 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. In particular, we aimed at identifying a suitable Datalog fragment with the following properties:

- 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.

Note that a mere *ad hoc* translation of CTL to Datalog (as outlined in Section 2) 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.

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

Results. 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. The program of Example 2, for instance, is a Datalog LIT program. On the other hand, the rule $Txy \leftarrow Exz, Tzy$ is not guarded.

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 2, Theorem 10), the simple CTL formula $\mathbf{AF}\varphi$ 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 *generalized literals* 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 2.) 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.

We prove the following relevant facts about Datalog LITE:

- For fixed Datalog LITE programs model-checking is feasible in linear time (Theorem 19). This remains true even for variable Datalog LITE programs whose predicate-arities are bounded by a constant or whose guards are all input atoms. We develop and describe a linear-time model checking algorithm in Section 2. It is interesting to note that this algorithm does not use Tarjan's linear algorithm for identifying strongly connected components, but rather relies on well-known linear-time algorithms for propositional Horn Clauses. Thus, our algorithm is based on a linear time solvable fragment of the propositional satifsfiability problem; note that related procdures like the Davis Putnam procedure have also found applications in model checking [Biere et al., 1999].
- 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.

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

- 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 20).
- 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.
- 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. These subfragments will be defined in Section 6.

• 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. Further Datalog LITE can define new modalities, e.g. via Boolean combinations of existing ones.

Related work 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 [Ramakrishnan et al., 1997, Cui et al., 1998, Charatonik and Podelski, 1998]. In a recent paper by Immerman and Vardi [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 as introduced in the present paper.

2 From Datalog to Datalog LIT

Definition 3 A *Datalog rule* is an expression of the form $H \leftarrow B_1, \ldots, B_r$ 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.

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].

Definition 4 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 per 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 5 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 6 *On input of two binary relations E, F and one unary relation S the following Datalog LIT query (Π, T) computes the set of all elements from which an element outside S is reachable along a path, on which edges from E and F have to alternate:*

$$Tx \leftarrow \neg Sx \qquad Ux \leftarrow Fxy, Ty \qquad Tx \leftarrow Exy, Uy$$

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 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, 1990b]), 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*

(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.

We defer the proof to section 4, 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 *Over finite structures, the well-foundedness query cannot be expressed by stratified Datalog programs with only monadic head predicates.*

Theorem 11 *Over Kripke structures, every Datalog LIT query with monadic output predicate is equivalent to a Datalog LIT query where all head predicates are monadic.*

Corollary 12 *Datalog LIT does not express the well-foundedness query.*

Remark: Since CTL expresses well-foundedness, 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 13 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 14 *Every query in stratified Datalog is equivalent to some formula in $\bigcup_{k<\omega} L(\Sigma_k)$.*

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

Theorem 15 *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.*

3 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 16 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 $(\forall \bar{y}.Rx\bar{y})Sx\bar{y}$, 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 5 is applicable to rules with generalized literals.

Definition 17 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.

Remarks.

(i) Datalog LITE is indeed stronger than Datalog LIT, since the program $Wx \leftarrow (\forall y.Eyz)Wy$ computes the well-foundedness query on both finite and infinite structures. Alternatively, Datalog LIT and Datalog LITE can be separated by the fact that the GAME problem [Abiteboul et al., 1995] is expressible in Datalog LITE by a single rule $Wx \leftarrow Exy, (\forall z.Eyz)Wz$, but not expressible by stratified programs [Dahlhaus, 1987, Kolaitis, 1990]. Our results however are tighter, since well-foundedness separates Datalog LITE from Datalog LIT, but is computable in stratified Datalog.

(ii) Note that according to Definition 5, every Datalog LIT rule contains a guard, while by Definition 16, a generalized literal also contains a guard. It will become clear in section 7 that both kinds of guards correspond to a single concept in the guarded fragment of first-order logic.

(iii) 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 18 *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 19 *The complexity results about Datalog LIT (Theorem 8) hold also for Datalog LITE.*

We shall see later (cf. Lemma 22) 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 to write programs in a much more compact way. By the following proposition, the exponential time bound for this case is indeed optimal:

Proposition 20 *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.*

4 Proof of Linear Time Algorithms and Elimination of Non-Input Guards

This section outlines the proof of Theorems 8 and 19. Recall that we have to show that for Datalog LITE, the complexity of evaluating a query 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.

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 $\mathfrak{A}$. The *distance* of two elements and 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. The following lemma is an immediate observation.

Lemma 21 *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. From this observation, one easily obtains non-expressibility results: for example, transitive closures cannot be expressed in Datalog LITE. The most important application of locality invariance is the following lemma:

Lemma 22 *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*

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

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

Proof.(Idea) 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 with this 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. □

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 [Itai and Makowsky, 1987, Dowling and Gallier, 1984, 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 [Berman et al., 1995].

Algorithm EvalLIT. For Datalog LIT, 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 $Fx \leftarrow Gx, M_y$, and $M_y \leftarrow My$. 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 $|\mathfrak{A}|$.
(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.

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

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

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
  for all tuples $\bar{c}\bar{d} \in \alpha^A$
    if HCB H_0 **contains a rule** r **for** $R_\varphi \bar{d}$
      then add $\beta\bar{c}\bar{d}$ **to the body of** r
      else set $r := R_\varphi \bar{d} \leftarrow \beta\bar{c}\bar{d}$ **and add** r **to** H_0
for all rules $r \in \Pi$
  replace generalized literals φ **in** r **by** R_φ
call EvalLIT$(\Pi, \mathfrak{A})$

Algorithm EvalLITE. For Datalog LITE, we shall adapt the proof of Theorem 8 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. Thus, **EvalLIT** is extended as follows.

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 [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 $|\mathfrak{A}|$.)

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.

5 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]. The formulae of the logics we describe here are evaluated on Kripke structures (with one or more transition relations) at a particular state. 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 $\mathcal{K} = (S, (E_a)_{a\in A}, P_1, P_2, \dots)$ where for a finite set A of 'actions', and each $a \in A$ we have a binary relation E_a on S, and where the P_i are monadic relations representing the atomic propositions.

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, $\mathcal{K} = (S, (E_a)_{a \in A}, P_1, P_2, \dots)$ 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 on in state formulae. The state formulae are defined by the rules (S1) and (S2) (read everywhere state formula instaed 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 $\bar{s} = s_0, s_1, s_2, \dots$ 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.

(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 $X, Y, \dots$) 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.

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 propositinal variable X defines on every Kripke structure $\mathcal{K}$ (with state set S) an operator $\psi^{\mathcal{K}}$: $\text{Pot}(S) \to \text{Pot}(S)$ 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. 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: alternation-free μ-calculus and CTL. 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 $\mu X.\varphi \vee (\psi \wedge \Box X)$.

6 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 (cf. [Marx et al., 1996]), 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:

$$
\begin{aligned}
Hx &\leftarrow (\neg)P_1x, \ldots, (\neg)P_rx \\
Hx &\leftarrow Exy, (\neg)P_1y, \ldots, (\neg)P_ry \\
Hx &\leftarrow (\forall y.Exy)P_0y, (\neg)P_1y, \ldots, (\neg)P_ry
\end{aligned}
$$

Further it is obvious that every modal program is equivalent to a program in reduced form where all rules are of one of the forms

$$
\begin{aligned}
Hx &\leftarrow (\neg)Px, (\neg)Qx \\
Hx &\leftarrow Exy, Py \\
Hx &\leftarrow (\forall y.Exy)Py
\end{aligned}
$$

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

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

$$
\begin{aligned}
&Hx \leftarrow Px, Exy, Hy \\
\text{or } &Hx \leftarrow Px, (\forall y \,.\, Exy)Hy.
\end{aligned}
$$

(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

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 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 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, 1999]: (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, 1999] 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, 1995]. 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.

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

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

The translation from fixed-point sentences into Datalog LITE programs is computable in linear time.

Corollary 33 *Alternation-free* μGF *admits linear time model checking.*

8 Conclusion

In summary, Datalog LITE is a robust fragment 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 (the description of) a finite state system is stored in a relational database.

The following table summarizes our results about the expressive power of Datalog LITE:

Deductive Language	*Logic*
Datalog LITE	alternation-free guarded fixed point logic
modal Datalog LITE	alternation-free μ-calculus
recursion-free Datalog LITE	guarded fragment
modal recursion-free Datalog LITE	multimodal logic

There are several interesting questions related to Datalog LITE we are currently studying. One is to find suitable extension of this language to express CTL* and LTL. It seems that this can be done by adding very few new primitives along the lines of a general approach to extending Datalog [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.

References

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Andréka et al., 1998] 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 et al., 1988] Apt, K. R., Blair, H. A., and Walker, A. (1988). Towards a theory of declarative knowledge. In *Foundations of DD and LP*, pages 89–148.

[Bancilhon et al., 1986] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In *PODS'86*, pages 1–16.

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

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without BDDs. Submitted.

[Ceri et al., 1990] Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Databases*. Surveys in Computer Science. Springer.

[Charatonik and Podelski, 1998] Charatonik, W. and Podelski, A. (1998). Set-based analysis of reactive infinite-state systems. In *TACAS'98*, LNCS. Springer.

[Clarke Jr. et al., 1986] Clarke Jr., E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263.

[Cleaveland, 1993] Cleaveland, R. (1993). A linear time model checking algorithm for the alternation free modal μ-calculus. *Formal Methods in System Design*, 2:121–147.

[Cui et al., 1998] 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*, volume 1490 of *LNCS*, pages 1–20. Springer.

[Dahlhaus, 1987] Dahlhaus, E. (1987). Skolem normal forms concerning the least fixpoint. In *Computation Theory and Logic*, volume 270 of *LNCS*, pages 101–106. Springer.

[Dowling and Gallier, 1984] 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 and Flum, 1995] Ebbinghaus, H.-D. and Flum, J. (1995). *Finite model theory*. Perspectives in Mathematical Logic. Springer-Verlag.

[Eiter et al., 1997] Eiter, T., Gottlob, G., and Veith, H. (1997). Modular logic programming and generalized quantifiers. In *LPNMR'97*, volume 1265 of *LNCS*, pages 290–309. Springer.

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

[Gelder et al., 1991] 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 et al., 1998] Gottlob, G., Grädel, E., and Veith, H. (1998). Datalog LITE: deductive versus temporal reasoning in verification. Technical Report DBAI-TR-98-22, Institut für Informationssysteme, TU Wien.

[Grädel, 1999] Grädel, E. (1999). On the restraining power of guards. *J. Symb.Logic*. To appear.

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

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

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

[Kolaitis, 1990] Kolaitis, P. G. (1990). Implicit definability on finite structures and unamboguous computations. *Inf.Comp.*, 90:50–66.

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

[Marx et al., 1996] Marx, M., Pólos, L., and Masuch, M., editors (1996). *Arrow Logic and Multi-Modal Logic*. CSL Publications.

[Minoux, 1988] 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 and Shapiro, 1990] Moret, B. and Shapiro, H. (1990). *Algorithms from P to NP*. Benjamin/Cummings.

[Murakami, 1990] 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 et al., 1997] 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*, volume 1254 of *LNCS*, pages 143–154. Springer.

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

[van Emde Boas, 1990a] van Emde Boas, P. (1990a). Machine models and simulation. In van Leeuven, J., editor, *Handbook of Theoretical Computer Science*, volume A, pages 3–66. Elsevier.

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

[Vardi, 1998] Vardi, M. Y. (1998). Reasoning about the past with two-way automata. In *ICALP 1998*, LNCS 1443, pages 628–641. Springer.