Logic Programming Approach to Automata-Based Decision Procedures

Gulay Unel and David Toman

D.R. Cheriton School of Computer Science, University of Waterloo
{gunel,david}@cs.uwaterloo.ca

Abstract. We propose a novel technique that maps decision problems in WS1S (weak monadic second-order logic with n successors) to the problem of query evaluation of Complex-value Datalog queries. We then show how the use of advanced implementation techniques for Logic Programs, in particular the use of tabling in the XSB system, yields a considerable improvement in performance over more traditional approaches. We also explore various optimizations of the proposed technique based on variants of tabling and goal reordering. Although our primary focus is on WS1S, it is straightforward to adapt our approach for other logics with existing automata-theoretic decision procedures, for example WS2S.

1 Introduction

Monadic second-order logics provide means to specify regular properties of systems in a succinct way. In addition, these logics are decidable by the virtue of the connection to automata theory. However, only recently tools based on these ideas—in particular the MONA system [17]—have been developed and shown to be efficient enough for practical applications [15].

However, for reasoning in large theories consisting of relatively simple constraints, such as theories capturing UML class diagrams or database schemata, the MONA system runs into a serious state-space explosion problem—the size of the automaton capturing the (language of) models for a given formula quickly exceeds the space available in most computers. Surprisingly, the problem can be traced to the *automata product* operation that is used to translate conjunction in the original formulæ rather than to the projection/determinization operations needed to handle quantifier alternations.

This paper introduces a technique that combats this problem. However, unlike most other approaches that usually attempt to use various compact representation techniques for automata, e.g., based on BDDs [5] or on state space factoring using a *guided* automaton [17], our approach is based on techniques developed for program evaluation in deductive databases, in particular on the *Magic Set transformation* [2] and *SLG resolution*, a top-down resolution-based approach augmented with memoing [9,10]. We also study the impact of using other optimization techniques developed for Logic Programs, such as goal reordering.

The main contribution of the paper is establishing the connection between the automata-based decision procedures for WS1S (and, analogously, for WS2S)

V. Dahl and I. Niemelä (Eds.): ICLP 2007, LNCS 4670, pp. 165–179, 2007.

and query evaluation in Complex-value Datalog (Datalog^{cv}). Indeed, the complexity of query evaluation in Datalog^{cv} matches the complexity of the WS1S decision procedure and thus it seems like an appropriate tool for this task. Our approach is based on representing automata using nested relations and on defining the necessary automata-theoretic operations using Datalog^{cv} programs. This reduces to posing a closed Datalog^{cv} goal over a Datalog^{cv} program representing implicitly the final automaton. This observation combined with powerful program execution techniques developed for deductive databases, such as the Magic Set rewriting and SLG resolution, limit the explored state space to elements needed to show non-emptiness of the automaton and, in turn, satisfiability of the corresponding formula.

In addition to showing the connection between the automata-based decision procedures and query evaluation in Datalog^{cv}, we have also conducted experiments with the XSB [27] system that demonstrate the benefits of the proposed method over more standard approaches.

The remainder of the paper is organized as follows. In Section 2 we formally introduce monadic second-order logic and the connection to finite automata. We also define Datalog^{cv} programs, state their computational properties, and briefly discuss techniques used for program evaluation. Section 3 shows how Datalog^{cv} programs can be used to implicitly represent a finite automaton and to implement automata-theoretic operations on such a representation. In this paper we only present results for the WS1S logic. However, the results extend immediately to WS2S. Experimental results for the proposed methods are presented in Section 4. Related work is discussed in Section 5. Finally, conclusions and future research directions are given in Section 6.

2 Background and Definitions

In this section we provide definitions needed for the technical development in the rest of the paper.

2.1 Weak Second-Order Logic (WS1S)

First, we define the syntax and semantics of the weak second-order logic of one successor and comment on techniques used to show its decidability.

Definition 1. *The formulas of second-order logics are defined as follows.*

- *the expressions $s(x, y)$, $x \subseteq y$ for x, y second-order variables are atomic formulas, and*
- *given formulas φ and ϕ and a variable x, the expressions $\varphi \wedge \phi$, $\neg\varphi$, and $\exists x : \varphi$ are also formulas.*

Additional common syntactic features can be defined by the following. Variables for individuals (first-order variables) can be simulated using second-order variables bound to singleton sets; a property expressible in WS1S. Thus we allow

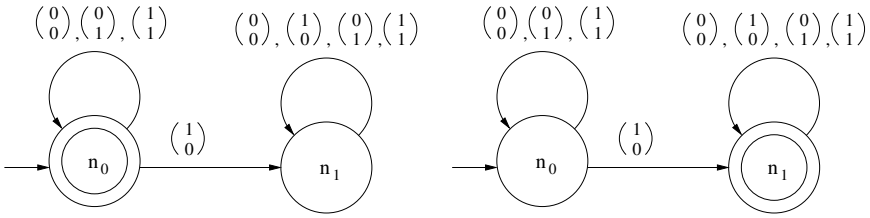


Fig. 1. Automata representing the formulae $x \subseteq y$ and $\neg(x \subseteq y)$

$x \in y$ for $x \subseteq y$ whenever we know that x is a singleton. We also use the standard abbreviations $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, and $\forall x : \varphi$ for $\neg\exists x : \neg\varphi$.

The semantics of WS1S is defined w.r.t. the set of natural numbers (successors of 0); second-order variables are interpreted as finite sets of natural numbers. The interpretation of the atomic formula $s(x, y)$ is fixed to relating singleton sets $\{n + 1\}$ and $\{n\}$, $n \in \mathbf{N}$.[1] Truth and satisfiability of formulas is defined with the help of valuations mapping variables to finite sets of natural numbers in a standard way.

Connection to Finite Automata. The crux of the connection lies in an observation that, for every WS1S formula, there is an automaton that accepts exactly the (string representations of) models of a given formula [28]. Since each variable of WS1S is interpreted by a finite set of natural numbers, such an interpretation can be captured by a finite string. Satisfying interpretations of formulas (with k free variables) can be represented as sets of strings over $\{0, 1\}^k$. The i-th component corresponds to the interpretation of the i-th variable and is called a *track*. It turns out that sets of the above strings form regular languages and thus can be recognized using an automaton. Satisfiability then reduces to checking for non-emptiness of the language accepted by such an automaton.

Definition 2. *A finite automaton is a 5-tuple $A = (N, X, S, T, F)$, where N is the set of states (nodes), X is the alphabet, S is the initial (starting) state, $T \subseteq N \times N \times X$ is the transition relation, and F is the set of final states.*

Given a WS1S formula φ, the automaton A_φ can be effectively constructed starting from automata for atomic formulæ using automata-theoretic operations.

Proposition 1. *Let φ be a WS1S formula. Then there is an automaton A_φ such that φ is satisfiable if and only if $L(A_\varphi) \neq \emptyset$, where $L(A)$ is the language accepted by A.*

Example 1. The automaton A_φ for the formula $\varphi = x \subseteq y$ is shown in the left part of Figure 1, the complement automaton $A_{\neg\varphi}$ that represents $\neg\varphi = \neg(x \subseteq y)$ is shown in the right part of Figure 1. The labels on the edges are elements of

[1] The atomic formula $s(x, y)$ is often written as $x = s(y)$ in literature, emphasizing its nature as a *successor* function.

the alphabet of the automaton and capture the valuations of variables allowed for a particular transition. The tracks in the strings accepted by the automata represents the valuation for the variables x (first track), and y (second track).

Similarly, the automaton $A_{\varphi\wedge\phi}$ is the product automaton of A_φ and A_ϕ and accepts $L(A_\varphi) \cap L(A_\phi)$, the satisfying interpretations of $\varphi \wedge \phi$. The automaton $A_{\exists x:\varphi}$, the projection automaton of A_φ, accepts satisfying interpretations of $\exists x : \varphi$. Intuitively, the automaton $A_{\exists x:\varphi}$ acts as the automaton A_φ for φ except that it is allowed to guess the bits on the track of the variable x. We give the actual algorithms for the inductive construction of A_φ in the following section (in a Datalog$^{\text{cv}}$ syntax). While checking for emptiness can be done in time polynomial in the size of an automaton, the size of A_φ is non-elementary in the size of φ (more precisely, in the depth of quantifier alternation of φ). This bound is tight for WS1S decision problem yielding an overall non-elementary decision procedure.

2.2 Datalog for Complex Values

In the remainder of this section we define a query language that serves as the target of our approach to WS1S decision procedure.

Complex Data Model. The complex-value data model is an extension of the standard relational model that allows tuples and finite sets to serve as values in the place of atomic values [1]. Each value is assigned a finite *type* generated by the type grammar "$\tau := \iota \mid [\tau_1, \ldots, \tau_k] \mid \{\tau\}$", where ι stands for the type of uninterpreted atomic constants, $[\tau_1, \ldots, \tau_k]$ for a k-tuple consisting of values belonging to the types $\tau_1, \ldots, \tau_k$, respectively, and $\{\tau\}$ for a finite set of values of type τ. Relations are interpreted as sets of values of a given type[2]. The model is equipped with several *built-in* relations, e.g., the equality $=$ (extended to all types), the subset relation $\subseteq$ (defined for set types), the tuple constructor (that relates tuples of values to the individual values), the singleton set constructor (relating values of a type to singleton sets of the appropriate set type), etc.

Complex-value Queries. The extended data model induces extensions to relational query languages and leads to the definition of *complex-value relational calculus* (calc$^{\text{cv}}$) and a deductive language for complex values, Datalog$^{\text{cv}}$—the language of Horn clauses built from literals whose arguments range over complex-valued variables and constants [2,26]. Datalog$^{\text{cv}}$ programs and queries are defined as follows:

Definition 3. *A* Datalog$^{\text{cv}}$ atom *is a predicate symbol with variables or complex-value constants as arguments.*

A Datalog$^{\text{cv}}$ database (program) *is a finite collection of Horn clauses of the form $h \leftarrow g_1, \ldots, g_k$, where h (called head) is an atom with an optional* grouping specification *and $g_1, \ldots, g_k$ (called goals) are literals (atoms or their negations).*

[2] We use relations of arity higher than one as a shorthand for sets—unary relations—of tuples of the same arity.

The grouping is syntactically indicated by enclosing the grouped *argument in the $\langle\cdot\rangle$ constructor; the values then range over the set type of the original argument.*

We require that in every occurrence of an atom the corresponding arguments have the same finite type and that the clauses are stratified with respect to negation.

A Datalog$^{\text{cv}}$ *query is a clause of the form $\leftarrow g_1, \ldots, g_k$.*

Evaluation of a Datalog$^{\text{cv}}$ *query (with respect to a* Datalog$^{\text{cv}}$ *database P) determines whether $P \models g_1, \ldots, g_k$.*

Datalog$^{\text{cv}}$ is equivalent to the complex-value calculus in expressive power [1]. However, the ability to express transitive closure without resorting to the powerset construction aids our goal of using Datalog$^{\text{cv}}$ to represent finite automata and to test for emptiness.

Proposition 2. *The complexity of* Datalog$^{\text{cv}}$ *query evaluation is non-elementary.*

Note that the complexity matches that of the decision procedures for WS1S and thus mapping of WS1S formulas to Datalog$^{\text{cv}}$ queries can be done efficiently.

To simplify the notation in the following we allow terms constructed of constants, variables, and finite number of applications of tuple and set constructors to appear as arguments of atoms. For example $p(\{x\}, y) \leftarrow q([x, y])$ is a shorthand for $p(z, y) \leftarrow q(w), w = [x, y], z = \{x\}$, where $w = [x, y]$ is an instance of a tuple constructor and $z = \{x\}$ of a set constructor built-in relations as discussed in our overview of the complex-value data model.

Evaluation of Datalog$^{\text{cv}}$ **Programs.** The basic technique for evaluation of Datalog$^{\text{cv}}$ programs is commonly based on a fixed-point construction of the minimal Herbrand model (for Datalog$^{\text{cv}}$ programs with *stratified negation* the model is constructed w.r.t. the stratification) and then testing whether a ground (instance of the) query is contained in the model. The type restrictions guarantee that the fixpoint iteration terminates after finitely many steps. While the naive fixed-point computation can be implemented directly, efficient query evaluation engines use more involved techniques such as the semi-naive evaluation, goal/join ordering, etc. In addition, whenever the query is known as part of the input, techniques that allow constructing only the relevant parts of the minimal Herbrand model have been developed. Among these the most prominent are the magic set rewriting (followed by subsequent fixed-point evaluation) and the top-down resolution with memoing—the SLG resolution.

Magic Sets. The main idea behind this approach is to restrict the values derived by a fixpoint computation to those that can potentially aid answering a given query. This is achieved by program transformation based on adding *magic predicates* to clauses that limit the breadth of the fixpoint computation at each step. These predicates are *seeded* by the values in the query (as those are the only ones the user desires to derive); more values are added to the interpretations of the magic predicates by means of additional clauses that *relax* the limit depending on what additional subqueries for a particular predicate need to be asked

to answer the original query. This process then becomes a part of the fixpoint evaluation itself.

SLG Resolution. In contrast, the SLG resolution is based on the more common SLD resolution used in PROLOG systems. The difference is in memoing what subgoals have been already resolved against and what answer substitutions were obtained, if any. Thus, whenever a more specific goal is to be selected, the memoing information is inspected to prevent repetitive computation. This mechanism also prevents infinite resolution paths in evaluation of Datalog$^{\text{cv}}$ queries. There are efficient scheduling strategies implemented for tabled logic programs:

- Batched Scheduling: provides space and time reduction over the naive strategy which is called single stack scheduling.
- Local Scheduling: provides speedups for programs that require answer subsumption.

Surprisingly, it can be shown that both of the techniques essentially simulate each other and that the magic predicates match the memoing data structures (modulo open terms). For detailed description of the above techniques see [3,20] and [27], respectively.

3 Automata and Datalog for Complex Values

In this section we present the main contribution of our approach: Given a WS1S formula φ we create a Datalog$^{\text{cv}}$ program P_φ such that an answer to a reachability/transitive closure goal w.r.t. this program proves satisfiability of φ.

However, we do not attempt to map the formula φ itself to Datalog$^{\text{cv}}$. Rather, we represent the construction of A_φ—the finite automaton that captures models of φ—as a Datalog$^{\text{cv}}$ program P_φ. This enables the use of the efficient evaluation techniques discussed in Section 2.2.

3.1 Representation of Automata

First, we fix the representation for automata that capture models of WS1S formulæ. Given a WS1S formula φ with free variables $x_1, \ldots, x_k$ we define a Datalog$^{\text{cv}}$ program P_φ that defines the following predicates:

1. $\mathsf{Node}_\varphi(n)$ representing the nodes of A_φ,
2. $\mathsf{Start}_\varphi(n)$ representing the starting state,
3. $\mathsf{Final}_\varphi(n)$ representing the set of final states, and
4. $\mathsf{Trans}_\varphi(nf_1, nt_1, \overline{x})$ representing the transition relation.

where $\overline{x} = \{x_1, x_2, \ldots, x_k\} \in X_\varphi$ is the set of free variables of φ; concatenation of their binary valuations represents a letter of A_φ's alphabet.

Example 2. The following program P_φ represents the automaton A_φ shown in the left part of Figure 1:

$$\begin{array}{lll}
\mathsf{Node}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 0, 0) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 0, 0) \leftarrow \\
\mathsf{Node}_\varphi(n_1) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 0, 1) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 1, 0) \leftarrow \\
\mathsf{Start}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_0, 1, 1) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 0, 1) \leftarrow \\
\mathsf{Final}_\varphi(n_0) \leftarrow & \mathsf{Trans}_\varphi(n_0, n_1, 1, 0) \leftarrow & \mathsf{Trans}_\varphi(n_1, n_1, 1, 1) \leftarrow
\end{array}$$

Note that while for atomic formulas, the values representing nodes are atomic, for automata corresponding to complex formulæ these values become complex.

3.2 Automata-Theoretic Operations

We define the appropriate automata-theoretic operations: negation, conjunction, projection, and determinization used in decision procedures for the logics under consideration as programs in Datalog$^{\mathrm{cv}}$ as follows.

Definition 4. *The program $P_{\neg\alpha}$ consists of the following clauses added to the program P_α:*

1. $\mathsf{Node}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)$
2. $\mathsf{Start}_{\neg\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)$
3. $\mathsf{Final}_{\neg\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n), \neg\mathsf{Final}_\alpha(n)$
4. $\mathsf{Trans}_{\neg\alpha}(nf_1, nt_1, \overline{x}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x})$

The following lemma is immediate:

Lemma 1. *If P_α represents A_α then $P_{\neg\alpha}$ represents $A_{\neg\alpha}$.*

The conjunction automaton which represents the conjunction of the two formulæ that original automata represent is defined as follows.

Definition 5. *The program $P_{\alpha_1\wedge\alpha_2}$ consists of the union of programs P_{α_1} and P_{α_2} and the following clauses*

1. $\mathsf{Node}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Node}_{\alpha_1}(n_1), \mathsf{Node}_{\alpha_2}(n_2)$
2. $\mathsf{Start}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Start}_{\alpha_1}(n_1), \mathsf{Start}_{\alpha_2}(n_2)$
3. $\mathsf{Final}_{\alpha_1\wedge\alpha_2}([n_1, n_2]) \leftarrow \mathsf{Final}_{\alpha_1}(n_1), \mathsf{Final}_{\alpha_2}(n_2)$
4. $\mathsf{Trans}_{\alpha_1\wedge\alpha_2}([nf_1, nf_2], [nt_1, nt_2], \overline{x}, \overline{y}, \overline{z}) \leftarrow$
   $\mathsf{Trans}_{\alpha_1}(nf_1, nt_1, \overline{x}, \overline{y}),\ \mathsf{Trans}_{\alpha_2}(nf_2, nt_2, \overline{y}, \overline{z})$

The sets of variables $\overline{x}, \overline{y}$ represent the free variables of the formula A_{α_1} and $\overline{y}, \overline{z}$ of the formula A_{α_2}.

Again, immediately from the definition we have:

Lemma 2. *Let P_{α_1} represent A_{α_1} and P_{α_2} represent A_{α_2}. Then $P_{\alpha_1\wedge\alpha_2}$ represents $A_{\alpha_1\wedge\alpha_2}$.*

The projection automaton which represents the existential quantification of a given formula is defined as follows.

Definition 6. *The program $P^u_{\exists\overline{x}:\alpha}$ is defined as the union of P_α with the clauses*

1. $\mathsf{Node}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Node}_\alpha(n)$
2. $\mathsf{Start}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Start}_\alpha(n)$

3. $\mathsf{Final}^u_{\exists\overline{x}:\alpha}(n) \leftarrow \mathsf{Final}_\alpha(n)$
   $\mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_0) \leftarrow \mathsf{Trans}_\alpha(n_0, n_1, \overline{x}, \overline{o}), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n_1)$
4. $\mathsf{Trans}^u_{\exists\overline{x}:\alpha}(nf_1, nt_1, \overline{y}) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_1, \overline{x}, \overline{y})$

The sets of variables $\overline{y}$ and $\overline{x}$ represent the free variables of the formula α, and $\overline{o} = \{0, 0, \ldots, 0\}$ where $|\overline{o}| = |\overline{y}|$.

Lemma 3. *If P_α represents A_α then $P^u_{\exists\overline{x}:\alpha}$ represents $A^u_{\exists\overline{x}:\alpha}$ which is nondeterministic automaton for the formula $\exists\overline{x} : \alpha$.*

The automaton obtained by the projection operation is nondeterministic. The following Datalog$^{\mathrm{cv}}$ program produces the representation of a deterministic automaton which accepts the same language as the nondeterministic one.

Definition 7. *The program $P_{\exists\overline{x}:\alpha}$ consists of the program $P^u_{\exists\overline{x}:\alpha}$ and the following clauses*

1. $\mathsf{Node}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Start}_{\exists\overline{x}:\alpha}(N)$
   $\mathsf{Node}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N_1), \mathsf{Trans}_{\exists\overline{x}:\alpha}(N_1, N, \overline{x})$
2. $\mathsf{Start}_{\exists\overline{x}:\alpha}(\{n\}) \leftarrow \mathsf{Start}^u_{\exists\overline{x}:\alpha}(n)$
3. $\mathsf{Final}_{\exists\overline{x}:\alpha}(N) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N), \mathsf{Final}^u_{\exists\overline{x}:\alpha}(n), n \in N$
4. $\mathsf{Trans}_{\exists\overline{x}:\alpha}(N_1, \langle n\rangle, \overline{x}) \leftarrow \mathsf{Node}_{\exists\overline{x}:\alpha}(N_1), \mathsf{Next}_{\exists\overline{x}:\alpha}(N_1, n, \overline{x})$
   $\mathsf{Next}_{\exists\overline{x}:\alpha}(N_1, n_2, \overline{x}) \leftarrow n_1 \in N_1, \mathsf{Trans}^u_{\exists\overline{x}:\alpha}(n_1, n_2, \overline{x})$

Lemma 4. *If $P^u_{\exists\overline{x}:\alpha}$ represents $A^u_{\exists\overline{x}:\alpha}$ then $P_{\exists\overline{x}:\alpha}$ represents a deterministic automaton $A_{\exists\overline{x}:\alpha}$.*

Last, the test for emptiness of an automaton has to be defined: To find out whether the language accepted by A_α is non-empty and thus whether α is satisfiable, a *reachability (transitive closure)* query is used.

Definition 8. *The following program TC_α computes the transitive closure of the transition function of A_α.*

1. $\mathsf{TransClos}_\alpha(n, n) \leftarrow$
2. $\mathsf{TransClos}_\alpha(nf_1, nt_1) \leftarrow \mathsf{Trans}_\alpha(nf_1, nt_2, \overline{x}), \mathsf{TransClos}_\alpha(nt_2, nt_1)$

Note that the use of magic sets and/or SLG resolution automatically transforms the transitive closure query into a reachability query.

Theorem 1. *Let φ be a WS1S (WS2S) formula. Then φ is satisfiable if and only if $P_\varphi, TC_\varphi \models \mathsf{Start}_\varphi(x), \mathsf{Final}_\varphi(y), \mathsf{TransClos}_\varphi(x, y)$.*

Example 3. Suppose that we have a formula $\exists y : y \subseteq x$, let A_ϕ be the automaton for the subformula $\phi = y \subseteq x$, we can use the following logic program to construct the automaton $A_{\exists y:\phi}$:

$\mathsf{Node}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Node}_\phi(n)$
$\mathsf{Start}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Start}_\phi(n)$
$\mathsf{Final}^u_{\exists y:\phi}(n) \leftarrow \mathsf{Final}_\phi(n)$
$\mathsf{Final}^u_{\exists y:\phi}(n_0) \leftarrow \mathsf{Trans}_\phi(n_0, n_1, 0, y), \mathsf{Final}^u_{\exists y:\phi}(n_1)$
$\mathsf{Trans}^u_{\exists y:\phi}(n_1, n_2, x) \leftarrow \mathsf{Trans}_\phi(n_1, n_2, x, y)$

This part computes the nondeterministic automaton ($A^u_{\exists y:\phi}$) representing the formula (see Definition 6).

$\mathsf{Node}_{\exists y:\phi}(N) \leftarrow \mathsf{Start}_{\exists y:\phi}(N)$
$\mathsf{Node}_{\exists y:\phi}(N) \leftarrow \mathsf{Node}_{\exists y:\phi}(N_1), \mathsf{Trans}_{\exists y:\phi}(N_1, N, x)$
$\mathsf{Start}_{\exists y:\phi}(\{n\}) \leftarrow \mathsf{Start}^u_{\exists y:\phi}(n)$
$\mathsf{Final}_{\exists y:\phi}(N) \leftarrow \mathsf{Node}_{\exists y:\phi}(N), \mathsf{Final}^u_{\exists y:\phi}(n), n \in N$
$\mathsf{Trans}_{\exists y:\phi}(N_1, \langle n \rangle, x) \leftarrow \mathsf{Node}_{\exists y:\phi}(N_1), \mathsf{Next}_{\exists y:\phi}(N_1, n, x)$
$\mathsf{Next}_{\exists y:\phi}(N_1, n_2, x) \leftarrow n_1 \in N_1, \mathsf{Trans}^u_{\exists y:\phi}(n_1, n_2, x)$
$\mathsf{TransClos}_{\exists y:\phi}(n, n) \leftarrow$
$\mathsf{TransClos}_{\exists y:\phi}(n_1, n_2) \leftarrow \mathsf{Trans}_{\exists y:\phi}(n_1, n_3, x), \mathsf{TransClos}_{\exists y:\phi}(n_3, n_2)$

This part computes the deterministic automaton ($A_{\exists y:\phi}$) representing the formula (see Definition 7), and the transitive closure of its transition function (see Definition 8). Note that determinization is not needed unless there is a negation operation after this step. The satisfiability query is:

$\leftarrow \mathsf{Start}_{\exists y:\phi}(n), \mathsf{Final}_{\exists y:\phi}(m), \mathsf{TransClos}_{\exists y:\phi}(n, m).$

The use of SLG resolution to evaluate the transitive closure goal allows us to construct only the relevant parts of the automaton in a goal-driven way:

Example 4. For the formula $\phi = \neg(x \in v) \vee \neg(\exists w : (y \in w) \wedge (z \in w))$ the bottom-up evaluation creates 240 transitions, and 16 transitive closure tuples for the starting node while the top-down evaluation with memoing technique creates only 1 transition, and 1 tuple in the transitive closure for the starting node as shown in Figure 2.

4 Experimental Evaluation

We compare the performance of the technique proposed in this paper and implemented using the XSB system[3] with the MONA system [15,17], one of the most advanced tools for reasoning in weak second-order logics (WS1S and WS2S).

The performance results for a set of formulas are given in Figures 3 and 4. We present a sample set of size 10 from the set of formulas we used in the experiments where #i represents a particular formula. The response times are measured in seconds; N/A means "Not Answered" in 120 seconds. The formulas are similar to the ones in T98 satisfiability test suite except we varied their sizes, the number of existential quantifiers, and free variables.

The results show that XSB outperforms MONA for the formulas with many free variables since it performs large numbers of conjunction operations very efficiently with the use of top-down query evaluation and pruning techniques. This can be easily traced to the effects of goal-driven evaluation of P_φ which become more pronounced for large theories consisting of relatively simple formulæ, such as those corresponding to constraints used in database schemata

[3] We simulate the set values in Datalog^{cv} by lists in XSB.

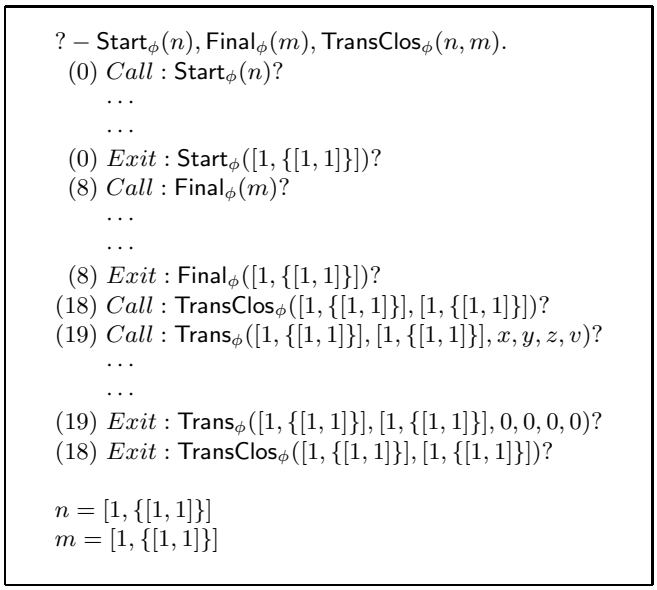


Fig. 2. Top-down evaluation of the program in Example 4

or UML diagrams. The experiments also compared different scheduling strategies of XSB namely the batched(XSB B) and the local(XSB L) ones. Batched scheduling performs better than local since our programs do not require answer subsumption. Experiments also show that tabling more predicates in addition to the auto-tabled ones (results in columns XSB B(T) and XSB L(T)) increases space requirements but enhances the performance substantially. The additional predicates we tabled are the Trans predicates in the programs that represent the determinization step. Since this step is critical in automaton construction, tabling the Trans predicate in addition to the Node predicate gives better results (see formulas 5, 9, 10).

On the other hand, MONA usually performs better on formulas that have less free variables and more quantifiers as it performs the projection operation faster than XSB. We believe that this is a practical problem caused by the implementation XSB uses for the evaluation of programs with nested relations and can be avoided using a more sophisticated implementation of Datalog^{cv}. In addition to this MONA uses a compact representation of automata based on BDDs [15,17,18] to enhance its performance, whereas XSB uses tries as the basis for tables combined with unification factoring [23,12]. The size of the trie structures is, in general, larger than the size of a corresponding BDD. However it is easier to insert tuples to a trie than into a BDD.

In the preliminary experiments [30] we conducted we also used CORAL, a deductive system that supports Datalog^{cv} and Magic sets. Our results showed that CORAL also performs better than MONA for the same formulas as XSB, however XSB is faster than CORAL in all cases.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
MONA	2.66	4.95	N/A	N/A	0.42	0.01	0.01	0.05	0.09	0.39
XSB B	0.01	0.01	0.11	0.01	35.72	1.74	N/A	0.01	6.02	94.64
XSB B(T)	0.01	0.01	0.01	0.01	15.88	0.18	N/A	0.01	0.29	10.96
XSB L	0.01	0.01	1.68	0.01	41.33	N/A	N/A	12.59	8.52	N/A
XSB L(T)	0.01	0.01	1.73	0.01	15.03	N/A	N/A	6.63	0.73	N/A

Fig. 3. Performance (secs) w.r.t. increasing number of quantifiers

	#7	#6	#8	#10	#5	#9	#1	#2	#3	#4
MONA	0.01	0.01	0.05	0.39	0.42	0.09	2.66	4.95	N/A	N/A
XSB B	N/A	1.74	0.01	94.64	35.72	6.02	0.01	0.01	0.11	0.01
XSB B(T)	N/A	0.18	0.01	10.96	15.88	0.29	0.01	0.01	0.01	0.01
XSB L	N/A	N/A	12.59	N/A	41.33	8.52	0.01	0.01	1.68	0.01
XSB L(T)	N/A	N/A	6.63	N/A	15.03	0.73	0.01	0.01	1.73	0.01

Fig. 4. Performance (secs) w.r.t. increasing number of variables

4.1 Heuristics for (Large) Conjunctions of Formulas

Representing theories that capture database schemas and/or UML diagrams often leads to large conjunctions of relatively simple formulas. Hence we develop heuristics that improve on the naive translation of a formula φ to a Datalog$^{\text{cv}}$ program P_φ presented in Section 3. Many of these heuristics are based on adapting existing optimization techniques for logic programs.

First, given a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ we have to decide which way the conjunctions should be associated (parenthesized). Figure 5 shows how performance depends on parenthesizing of a 4-way conjunction. In the experiment we test all permutations of two sets of 4 formulas w.r.t. all possible parenthesizations. The table reports the best and average times over all permutations for a given parenthesization. The results show that left associative parenthesizing is generally preferable.

To take advantage of the structure of the input conjunction, we propose another heuristics that produces a more appropriate goal ordering. We use a structure called a *formula graph*: the nodes of the graph G_φ are the conjuncts of φ and the edges connect formulas that share variables (the edge labels list the shared variables).

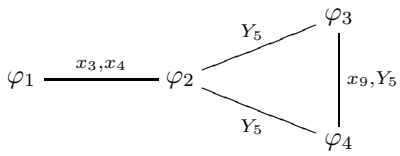
Example 5. Consider a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$ where:

$$\begin{aligned}
\varphi_1 &= \exists x_{10} : (((\exists Y_{30} : ((x_{10} \in Y_{30}) \wedge (x_4 \in Y_{30}))) \wedge (\exists Y_{40} : (x_{10} \in Y_{40}))) \\
&\quad \wedge \neg(\exists Y_{20} : ((x_{10} \in Y_{10}) \wedge (x_3 \in Y_{20})))) \\
\varphi_2 &= \neg((\exists x_1 : ((x_1 \in Y_5) \wedge (x_2 \in Y_5))) \wedge ((x_3 \in Y_5) \wedge (x_4 \in Y_5))) \\
\varphi_3 &= \neg(x_9 \in Y_5) \\
\varphi_4 &= (x_9 \in Y_5)
\end{aligned}$$

Formula	Parenthesizing	Best Time	Average Time	# Not Answered
1	$\varphi_i \wedge (\varphi_j \wedge (\varphi_k \wedge \varphi_l))$	0.56	23.23	5
	$((\varphi_i \wedge \varphi_j) \wedge (\varphi_k \wedge \varphi_l))$	0.77	7.00	4
	$((\varphi_i \wedge \varphi_j) \wedge \varphi_k) \wedge \varphi_l$	0.62	5.67	0
	$(\varphi_i \wedge (\varphi_j \wedge \varphi_k)) \wedge \varphi_l$	0.61	8.60	1
	$\varphi_i \wedge ((\varphi_j \wedge \varphi_k) \wedge \varphi_l)$	0.59	10.26	6
2	$\varphi_i \wedge (\varphi_j \wedge (\varphi_k \wedge \varphi_l))$	0.72	14.56	12
	$((\varphi_i \wedge \varphi_j) \wedge (\varphi_k \wedge \varphi_l))$	1.26	25.43	8
	$((\varphi_i \wedge \varphi_j) \wedge \varphi_k) \wedge \varphi_l$	0.94	24.18	2
	$(\varphi_i \wedge (\varphi_j \wedge \varphi_k)) \wedge \varphi_l$	1.65	27.00	5
	$\varphi_i \wedge ((\varphi_j \wedge \varphi_k) \wedge \varphi_l)$	0.74	18.96	11

Fig. 5. Performance (secs) Results w.r.t. Associativity

The formula graph for φ is as follows:



For this heuristic we also need to estimate size of the automaton A_φ. We use only very simple estimation rules for the automata operations:

- $|A_{\neg\varphi}| = |A_\varphi|$
- $|A_{\varphi_1 \wedge \varphi_2}| = |A_{\varphi_1}| \times |A_{\varphi_2}|$
- $|A_{\exists \overline{x}:\varphi}| = 2^{|A_\varphi|}$

The goal ordering heuristics for a formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ constructs a left-associative permutation as follows: it starts from the conjunct that has the largest estimated automaton and then finds its neighbor with the largest automaton (alternatively selecting another conjunct if there are no conjuncts left that satisfy this criteria). This step is repeated until all the conjuncts are processed. Intuitively in the case where a conjunction is applied on a large automaton and a small automaton, when top-down evaluation is used, for every final state we find in the first automaton we check all the final states of the second one and see if they form a final state in the conjunction automaton. Since we iterate on a small sized automaton in this case, ordering the formulas starting from the large ones is heuristically better. The experimental results shown in Figure 6 support this optimization. In the table *heuristic time* is the response time of the program for the rewriting generated by the proposed heuristics, *best time* is the fastest response time among all the programs generated for the formula, and similarly *worst time* is the slowest response time. The experiments show that in many cases the heuristic achieves a performance close to the performance of the program for the best possible ordering.

# of Conjuncts	Formula	Heuristic Time	Best Time	Worst Time
3	1	68.17	67.97	N/A
	2	68.45	68.45	N/A
	3	7.60	7.60	N/A
	4	94.46	1.04	N/A
	5	N/A	5.14	N/A
	6	0.42	0.42	3.18
4	1	1.06	0.56	N/A
	2	3.81	0.72	N/A
	3	0.66	0.64	N/A
5	1	12.61	0.94	N/A
	2	15.94	0.92	N/A
	3	2.6	0.50	N/A

Fig. 6. Performance (secs) results on ordering

5 Related Work

The connection between logic and automata was first considered by Büchi [6] and Elgot [13]. They have shown that monadic second-order logic over finite words and finite automata have the same expressive power, and we can transform formulas of this logic to finite automata and vice versa. Later, Büchi [7], McNaughton [19], and Rabin [22] proved that monadic second-order logic over infinite words (and trees) and finite automata also have the same expressive power. The practical use of this connection was investigated for temporal logics and fixed-point logics which led to the theory of model checking [4,32]. Another automata-theoretic construction was for μ-calculus [16,31] and could be used, in turn, for reasoning in expressive description logics. An extensive survey on automata and logic can be found in [28].

The logic-automaton connection has been used for implementing decision procedures for various logics. It is argued that the success of these procedures relies on efficient operations on a compact representation of automata based on BDDs [17,18].

We have used deductive techniques to represent and query nested-relational representation of finite automata. The systems used to test our implementation are CORAL [24,25,26] (in the preliminary experiments [30]), which provides the set-oriented data manipulation characteristics of relational systems, and XSB [27] (here sets have to be explicitly simulated). In terms of evaluation strategy, XSB uses top-down evaluation with memoing, whereas CORAL uses magic sets. The resolution technique XSB supports performs basic operations such as unification very efficiently also providing alternative scheduling strategies [14]. There are numerous other deductive systems which support logic-programming languages with sets and tuples, e.g., LDL [11,21]. There is also a BDD-based deductive database system called bddbddb [33] implemented to be used in program verification which translates Datalog programs into efficient BDD implementations. Hence, the techniques presented in this paper are complementary to BDDs.

Considerable work has been done on query optimization in relational and deductive database systems [20]. Query optimization in relational systems includes choosing join orders and cost models [8,29]. We use the ideas of SLG resolution [9,10], and magic sets rewriting [3] as deductive database optimization methods, and we are planning to use cost-based optimization methods to improve our query evaluation.

6 Conclusions and Future Work

In this paper, we presented a translation technique that maps satisfiability questions for formulas in WS1S to query answering in Datalog^{cv}. We have also demonstrated how evaluation techniques used for answering queries on these programs can provide efficient decision procedures for second-order logics. In addition we also study the impact of goal reordering and various other query optimization techniques on the performance of the decision procedure and propose heuristics for this purpose.

Future extensions of the proposed approach include extending the translation to other types of automata on infinite objects, e.g., to Rabin [22] and Alternating Automata [31], and on improving the upper complexity bounds by restricting the form of Datalog^{cv} programs generated by the translation (when used for decision problems in, e.g., EXPTIME). In all these cases, the goal is to match the optimal theoretical bounds while avoiding the worst-case behavior (inherent in most automata-based techniques) in as many situations as possible. We also consider the integration of the technique proposed in this paper with more standard techniques such as BDDs.

References

1. Abiteoul, S., Beeri, C.: The power of languages for the manipulation of complex values. VLDB Journal 4(4), 727–794 (1995)
2. Beeri, C., Naqvi, S., Shmueli, O., Tsur, a.S.: Set construction in a logic database language. JLP 10(3&4), 181–232 (1991)
3. Beeri, C., Ramakrishnan, R.: On the power of Magic. JLP 10(1/2/3&4), 255–299 (1991)
4. Bernholtz, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 142–155. Springer, Heidelberg (1994)
5. Bryant, R.E.: Symbolic boolean manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
6. Büchi, J.R.: Weak second-order arithmetic and finite automata. Z. Math. Logik Grundl. Math. 6, 66–92 (1960)
7. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. Int. Congr. for Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)
8. Chaudhuri, S.: An overview of query optimization in relational systems. In: PODS, pp. 34–43 (1998)
9. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. JLP 24(3), 161–199 (1995)
10. Chen, W., Warren, D.S.: Query evaluation under the well-founded semantics. In: PODS, pp. 168–179 (1993)

11. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., Tsur, S., Zaniolo, C.: The LDL system prototype. IEEE Trans. Knowl. Data Eng. 2(1), 76–90 (1990)
12. Dawson, S., Ramakrishnan, C.R., Skiena, S., Swift, T.: Principles and practice of unification factoring. TOPLAS 18(5), 528–563 (1996)
13. Elgot, C.C.: Decision problems of finite automata design and related arithmetics. Trans. Amer. Math. Soc. 98, 21–52 (1961)
14. Freire, J., Swift, T., Warren, D.S.: Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. Journal of Functional and Logic Programming 1998(3) (1998)
15. Henriksen, J.G., Jensen, J.L., Jörgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: MONA: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
16. Janin, D., Walukiewicz, I.: Automata for the modal μ-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)
17. Klarlund, N.: MONA & FIDO: The logic-automaton connection in practice. In: Computer Science Logic, pp. 311–326 (1997)
18. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. 13(4), 571–586 (2002)
19. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9, 521–530 (1966)
20. Mumick, I.S.: Query Optimization in Deductive and Relational Databases. PhD thesis, Department of Computer Science, Stanford University (1991)
21. Naqvi, S., Tsur, S.: A Logical Language for Data and Knowledge Bases. Computer Science Press (1989)
22. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. 141, 1–35 (1969)
23. Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., Warren, D.: Efficient tabling mechanisms for logic programs. In: Proc. ICLP, pp. 697–711 (1995)
24. Ramakrishnan, R., Bothner, P., Srivastava, D., Sudarshan, S.: CORAL - a database programming language. In: Workshop on Deductive Databases (1990)
25. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: Coral - control, relations and logic. In: Proc. VLDB Conference, pp. 238–250 (1992)
26. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL deductive system. VLDB Journal 3(2), 161–210 (1994)
27. Sagonas, K.F., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Proc. SIGMOD, pp. 442–453 (1994)
28. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, vol. 3, Springer, Heidelberg (1997)
29. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 1&2. Computer Science Press (1989)
30. Unel, G., Toman, D.: Deciding weak monadic second-order logics using complex-value datalog. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS(LNAI), vol. 3835, Springer, Heidelberg (2005)
31. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
32. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. LICS, pp. 322–331 (1986)
33. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. PLDI '04, pp. 131–144 (2004)