Bidirectional Answer Set Programs with Function Symbols[*]

Thomas Eiter and **Mantas Šimkus**
Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, Vienna, Austria
(eiter|simkus)@kr.tuwien.ac.at

Abstract

Current *Answer Set Programming* (ASP) solvers largely build on logic programming without function symbols. This limitation makes ASP decidable, but greatly complicates the modeling of indefinite time, recursive data structures (e.g., lists), and infinite processes and objects in general. Recent research thus aims at finding decidable fragments of ASP with function symbols and studying their complexity. We identify *bidirectional* ASP programs as an expressive such fragment that is useful, e.g., for reasoning about actions involving both the future and the past. We tightly characterize the computational complexity of bidirectional programs and of some of their subclasses, addressing the main reasoning tasks. Our results also imply that the recently introduced $\mathbb{FDNC}$ programs can be extended by *inverse predicates* while retaining decidability, but computational costs are unavoidably higher.

1 Introduction

In the last decade, *Answer Set Programming* (ASP) has become an important paradigm of Knowledge Representation and Reasoning [Baral, 2002]. ASP, with its roots in logic programming and databases, is based on logic programs with non-monotonic negation under the *answer set* semantics [Gelfond and Lifschitz, 1991], and is especially well-suited to represent problems involving common-sense reasoning. Supported by efficient reasoners,[1] ASP has been fruitfully applied in many areas of computer science and AI, including information integration, diagnosis, configuration management, reasoning about actions and change, etc. (see e.g. [Woltran, 2005] for a showcase and more details).

Current ASP implementations largely build on logic programming without function symbols. For this reason, the *answer sets* (also known as *stable models*) of an ASP program P are always finite relational structures over the relations and constants of P. This intrinsic finiteness is an acknowledged limitation in terms of expressiveness, and becomes apparent when modeling concepts that cannot be confined using a finite number of constants; this includes time, action sequences, recursive data structures like lists, and infinite processes and objects in general [Bonatti, 2004; Calimeri *et al.*, 2008].

Using function symbols, we can easily generate infinite domains, and so more compactly represent problems involving infinite objects. Unfortunately, an unrestricted use of function symbols makes ASP highly undecidable. In fact, already inference from Horn logic programs becomes undecidable, and equipped with negation under the answer set semantics, it is at the second level of the analytical hierarchy (as stable model existence is Σ_1^1-complete, cf. [Marek *et al.*, 1994]). This immense complexity, which discouraged many researchers, has two reasons. Firstly, function symbols make the Herbrand universe infinite, and the program can have infinitely many and possibly infinite answer sets. Secondly, each answer set of a program satisfies a minimality property that quantifies over (possibly infinitely many and infinite) interpretations.

The need to represent problems with infinite domains led several authors to identify fragments of ASP with function symbols that have lower complexity; the majority of them tailored conditions for the program dependency graph [Baselice *et al.*, 2007; Syrjänen, 2001; Bonatti, 2004; Calimeri *et al.*, 2008]. While many of the identified program classes are decidable, a weakness of this approach is that recognizing the programs from such classes may be difficult, if not undecidable. Moreover, many classes allow only finite models.

The recent $\mathbb{FDNC}$ programs [Simkus and Eiter, 2007], avoid the above limitations. Inspired by modal fragments of first-order logic where purely syntactic conditions yield decidability, $\mathbb{FDNC}$ is tailored to ensure tree-shaped stable models, and allow to inductively describe infinite objects. However, e.g., in reasoning about actions, they allow to talk naturally about the future, but not the past; in $\mathbb{FDNC}$ programs, atoms cannot be derived from structurally more complex atoms; e.g., in the famous Yale Shooting domain, a rule $\mathit{Dead}(x) \vee \mathit{Loaded}(x) \leftarrow \mathit{Dead}(\mathit{shoot}(x))$ is forbidden.

In this paper, we present *bidirectional (*BD*)-programs*, which are a close relative of $\mathbb{FDNC}$, but are relieved from the above restriction on atom dependency direction. This makes talking about both the *future* and the *past* possible. Furthermore, in contrast to $\mathbb{FDNC}$, it allows us to elegantly express eventuality conditions (A holds for some term) by backward propagation (which is not succinctly possible in $\mathbb{FDNC}$), and

[*]This work has been partially supported by the Austrian Science Fund (FWF) grants P20840 and P20841, and the EC project OntoRule (IST-2009-231875).

[1]See http://asparagus.cs.uni-potsdam.de/

to represent combinations of backward and forward reasoning; this makes BD-programs more expressive (and complex) than $\mathbb{FDNC}$ in general. Briefly, our main contributions are:

• We introduce the class of BD-programs with function symbols. Its syntactic restrictions, which modularly apply on the rules and are easy to test, ensure that the stable models of a program, like of $\mathbb{FDNC}$ programs, are tree-shaped. However, due to the minimality of stable models, bidirectionality of atom dependencies makes recognizing such models much more complicated, and needs special treatment.

• We thus provide a semantic characterization of stable models of BD-programs in terms of specially labeled trees. Based on it, we present algorithms for the basic reasoning tasks, including consistency testing, and brave/cautious entailment of ground/existential queries. The algorithms are different from those for $\mathbb{FDNC}$; we use automata-theoretic methods and, for disjunctive programs, a suitable automata acceptance notion to obtain optimal complexity results.

• We tightly characterize the complexity of BD-programs and some of its subclasses. Under bounded predicate arities, the aforementioned reasoning tasks are 2EXPTIME-complete for disjunctive BD-programs (hence provably harder than in $\mathbb{FDNC}$), but EXPTIME-complete if disjunction is disallowed; on the other hand, already brave reasoning from disjunctive BD-programs without negation is 2EXPTIME-hard. If only one function symbol is available (which suffices to model time), the complexity of reasoning drops to EXPSPACE in the disjunctive case, and to PSPACE if disjunction is disallowed.

The high expressivity of BD-programs makes them a possible host for encoding problems with matching complexity into ASP with function symbols. Examples are planning problems (e.g., conditional planning) or reasoning tasks in Description Logics (e.g., answering conjunctive queries in $\mathcal{SHIQ}$ and satisfiability testing in $\mathcal{SRIQ}$) that are 2EXPTIME-complete, or deciding datalog query containment (3EXPTIME-complete for pairs of non-recursive and recursive queries). To our knowledge, no ASP classes, as simple as BD-programs, with similar capacity were identified before.

2 Preliminaries

Answer Set Programming We assume a standard first-order language with distinct, countably infinite sets of *variables*, *constants*, and *function* and *predicate symbols* of positive arity. A *literal* is either an atom A (*positive literal*), or a negated atom $not\ A$ (*negative literal*).

A *disjunctive logic program* (briefly, *program*) is an arbitrary set of *(disjunctive) rules* r of the form

$$A_1 \vee \ldots \vee A_n \leftarrow L_1, \ldots, L_m, \qquad (1)$$

where $n+m>0$, all A_i are atoms (called *head atoms*) and all L_j are literals (called *body literals*). We let $\mathsf{head}(r) = \{A_1, \ldots, A_n\}$ and denote by $\mathsf{body}^+(r)$ (resp., $\mathsf{body}^-(r)$) the set of atoms that occur in positive (resp., negative) body literals of r. The rule r is a *fact*, if $m=0$; a *constraint*, if $n=0$; *normal*, if $n=1$; *positive*, if $\mathsf{body}^-(r)=\emptyset$ (we then let $\mathsf{body}(r) = \mathsf{body}^+(r)$); and *safe*, if all variables in r occur in $\mathsf{body}^+(r)$).

A program P is *positive* (*normal*, etc.), if all its rules are positive (normal, etc.). The *Herbrand universe* HU^P, *Herbrand base* HB^P and the grounding $\mathsf{Ground}(P)$ of P are as usual (cf. [Minker, 1988]). By $MM(P)$ we denote the set of $\subseteq$-minimal *(Herbrand) models* of P (viewing not as classical negation).

Recall that an interpretation I is a *stable model* of P, iff $I \in MM(P^I)$, where P^I is the *reduct* of P w.r.t. I [Gelfond and Lifschitz, 1991]. We denote by $SM(P)$ the set of all stable models of P. A program P is *consistent*, if $SM(P) \neq \emptyset$.

An *existential query* q is of the form $\exists\vec{x}.A(\vec{y})$, where A is an n-ary predicate symbol, $\vec{y}$ is an n-tuple of variables and constants, and $\vec{x}$ is a list of all variables in $\vec{y}$. If $\vec{x}$ is empty, then q is a *ground query*. A program P *bravely* (resp., *cautiously*) entails q, in symbols $P \models_b q$ (resp., $P \models_c q$), if some (resp., every) $I \in SM(P)$ contains a ground instance $A(\vec{t})$ of $A(\vec{x})$.

Alternating Tree Automata For an introduction to automata on infinite trees, we refer to [Thomas, 1990]. We here recall *2-way alternating tree automata* from [Vardi, 1998].

An *infinite tree* T is any prefix-closed set of words over the positive integers (denoted by $\mathbb{N}$), i.e., $T \subseteq \mathbb{N}^*$ such that $x{\cdot}c \in T$, where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, implies $x \in T$. T is *full* if, additionally, $x{\cdot}c' \in T$ for all $0<c'<c$. Each element $x \in T$ is a *node* of T, where ϵ (the empty word) is the root of T. The nodes $x{\cdot}c \in T$, where $c \in \mathbb{N}$, are the *successors* of x. By convention, $x{\cdot}0 = x$ and $(x{\cdot}i){\cdot}(-1) = x$ (note that $\epsilon{\cdot}(-1)$ is undefined). T is *k-ary* if it is full and each node in T has k successors. An *infinite path in* T is a prefix-closed node set $p \subseteq T$ such that for every $i \geq 0$ there is a unique $x \in p$ such that $|x| = i$. A *labeled tree* over an alphabet Σ is a tuple $(T, \mathcal{L})$, where $\mathcal{L} : T \to \Sigma$, i.e., a tree where the nodes are labeled with symbols from Σ.

For a set V of propositions, let $B(V)$ be the set of all Boolean formulas that can be built from $V \cup \{\mathbf{true}, \mathbf{false}\}$ using $\vee$ and $\wedge$ as connectives. We say that $I \subseteq V$ *satisfies* $\varphi \in B(V)$, if assigning $\mathbf{true}$ to each $p \in I$ and $\mathbf{false}$ to each $p \in V \setminus I$ makes φ true.

Let $[k] = \{-1, 0, 1, \ldots, k\}$. A *two-way alternating tree automaton (2ATA) over infinite k-ary trees* is a tuple $A = \langle \Sigma, Q, \delta, q_0, F\rangle$, where Σ is an input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to B([k] \times Q)$ is a transition function, $q_0 \in Q$ is an initial state, and F specifies an acceptance condition. We consider here *parity* acceptance, which is given by a chain F: $G_1 \subseteq G_2 \subseteq \ldots \subseteq G_m$ of subsets of Q where $G_m = Q$.

Informally, a *run* of a 2ATA A over a labeled tree $(T, \mathcal{L})$ is a tree T_r where each node $n \in T_r$ is labeled with $(x, q) \in T \times Q$. Here n describes a copy of A that is in state q and reads the node $x \in T$, and the labeling of its successor nodes must obey the transition function. Formally, a run (T_r, r) is labeled tree over $\Sigma_r = T \times Q$, which satisfies the following:

1. $\epsilon \in T_r$ and $r(\epsilon) = (\epsilon, q_0)$.
2. For each $y \in T_r$, with $r(y) = (x, q)$ and $\delta(q, \mathcal{L}(x)) = \varphi$, there is a set $S = \{(c_1, q_1), \ldots, (c_n, q_n)\} \subseteq [k] \times Q$ such that (i) S satisfies φ, and (ii) for all $1 \leq i \leq n$, we have that $y{\cdot}i \in T_r$, $x{\cdot}c_i$ is defined, and $r(y{\cdot}i) = (x{\cdot}c_i, q_i)$.

A run (T_r, r) is accepting, if every infinite path $p \subseteq T_r$ satisfies F as follows. Let $inf(p)$ be the set of states $q \in Q$ that occur infinitely often in p. Then p satisfies F, if an even i exists for which $inf(p) \cap G_i \neq \emptyset$ and $inf(p) \cap G_{i-1} = \emptyset$.

An automaton accepts a labeled tree, if there is a run that accepts it. By $L(A)$ we denote the set of trees that A accepts.

3 Bidirectional Programs

We now introduce *bidirectional* programs (BD-programs). After defining the full language, we focus on *core* programs, for which we give an algorithm for consistency checking. Other basic reasoning tasks and reasoning for the full language can be reduced to this problem.

BD-programs are syntactically restricted to ensure that stable models are tree-shaped. This allows us to recognize the possibly infinite stable models using a tree automaton.

Definition 3.1. (t-atom) An atom $R(t, s_1, \ldots, s_n)$ is called a t-*atom*, if each s_i is either a constant, or a variable that is distinct from and not occurring in t.

Example 3.1. Let x, y, z be variables, and c, d be constants. Then $R(x, c, d, y)$ is an x-atom, and $P(f(x), c, d, y)$ is an $f(x)$-atom. However, $R(g(x), z, x)$ is not a $g(x)$-atom because x is a subterm of $g(x)$ and x is occurs in the 3rd position of the atom. Also, $R(x, c, d, f(y))$ is not an x-atoms because functional terms are allowed in the first position only.

Definition 3.2. (BD-programs) Let c be a designated constant. A BD-*program* P consists of safe rules r of the form (1) such that, for some variable x, each atom in r is a t-atom where either $t = x$, $t = f(x)$ for some unary f, or $t = c$.

Example 3.2. E.g., rules $R(x, z, y) \leftarrow P(f(x), y, z)$ and $P(f(x), c, y) \leftarrow P(x, z, y), not\ R(g(x), y)$ are allowed in the syntax of BD-programs. However, $R(f(y), x) \leftarrow P(f(x), y)$ is not allowed since it does not fulfill the requirement above.

Observe that the rule in the introduction satisfies the above condition, and hence is allowed in BD-programs.

Note also that every atom in a stable model I of a BD-program P is of the form $R(t, c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are constants and t is of the form $f_n(\ldots f_1(c)\ldots)$. Observe also that the set of all terms $f_n(\ldots f_1(c)\ldots)$ forms a tree with root c, where a node $f(t)$ is a child of t. Thus, any I can be seen as a labeled tree in which each $R(t, c_1, \ldots, c_n) \in I$ is associated to the node t.

We consider next a program which is a safe variant of a program in [Baselice *et al.*, 2007], originally due to F. Fages.

Example 3.3. Let P consist of the rules

$$\begin{array}{llll} (1) & D(c) \leftarrow, & (3) & Q(x) \leftarrow Q(f(x)), \\ (2) & D(f(x)) \leftarrow D(x), & (4) & Q(x) \leftarrow D(x), not\ Q(f(x)). \end{array}$$

Then P is a BD-program, but is not finitely recursive according to [Baselice *et al.*, 2007]: indeed, due to (3), $Q(c)$ depends on infinitely many atoms $Q(f(c)), Q(f(f(c))), \ldots$

Observe that P is inconsistent. Indeed, any model I of P^I, by (1) and (2), must contain $D(t)$ for each ground term $t \in HU^P$. Then, by the rules (3) and (4), I must also contain $Q(t)$ for each ground term $t \in HU^P$. Hence, P^I contains only instances of rules (1)-(3), and thus I is not a minimal model of P^I.

If we replace (4) by $Q(x) \vee Q'(x) \leftarrow D(x),\ not\ Q(f(x))$, we obtain a consistent program with one stable model I_n for each natural number n: I_n consists of $Q(f^i(c))$ for each $i < n$, $Q'(f^i(c))$ for each $i \geq n$, and $D(f^i(c))$ for each $0 \leq i$.

To ease the development of algorithms, we focus on core programs over unary predicates, which are described next.

Definition 3.3. (Core programs) A BD-program P is a *core program*, if it consists of *core rules*, which have the form:

a) $A(c) \leftarrow$, where c is the special constant,

b) $A(f(x)) \leftarrow B(x)$, called f-*forward rule*,

c) $A(x) \leftarrow B(f(x))$, called f-*backward rule*, or

d) $A_1(x) \vee \ldots \vee A_m(x) \leftarrow [not]\ B_1(x), \ldots, [not]\ B_n(x)$, called *local rule* ($[not]$ stands for a possible negation).

Core programs are structurally simple, but as expressive as full BD-programs: by a structural transformation, we can reduce consistency of a BD-program P to consistency of a core program P'. First, by exploiting the restricted variable interaction (see Definition 3.2), P can be translated into a BD-program over unary predicates, arriving at rules with atoms of the form $A(x)$, $A(f(x))$, and $A(c)$ only (if predicate arities are bounded, the translation is polynomial). The resulting program can then be easily simulated by core rules.

Proposition 3.1. *Under bounded predicate arities, a disjunctive (resp., normal)* BD-*program P can be transformed in polynomial time into a disjunctive (resp., normal) core program P' such that P is consistent iff P' is consistent.*

We note that the translation moreover establishes a 1-1 correspondence between the stable models of P and P'.

4 Consistency in Normal BD-Programs

In this section, we develop an algorithm for testing consistency of normal core programs. To this end, we first introduce the notion of a *block tree*: a labeled tree that encodes a positive disjunction-free program with an interpretation for it. We then define *minimal* block trees, which are the ones where the encoded interpretation coincides with the least model of the encoded program, and define an automaton recognizing such trees. To decide consistency of normal core programs, we then focus on block trees where the encoded program equals the Gelfond-Lifschitz reduct w.r.t. the encoded interpretation.

4.1 Minimal Block Trees

(Minimal) block trees are trees labeled by *blocks* which intuitively consists of two parts: a set of predicate names and a set of rules associated to the node.

Definition 4.1. (Block) A *block* is any tuple $b = (\alpha, \mathcal{D}, \mathcal{R})$, where α is a constant or a function symbol, $\mathcal{D}$ is a set of unary predicates, and $\mathcal{R}$ is a set of positive disjunction-free core rules such that: (i) if α is a constant, then $\mathcal{R}$ has no f-backward rule for any function f, and (ii) if α is a function symbol f, then for each g-backward rule in $\mathcal{R}$ we have $g = f$. In case (i), b is a *root block*, and in case (ii) b is a *child block*.

Intuitively, a root block can be used as a root of the tree and a child block inside the tree. We next illustrate conditions (i-ii).

Example 4.1. Consider the following 3 blocks:

$b_1 = (c, \{A, C\}, \{A(x) \leftarrow ; B(f(x)) \leftarrow A(x); D(g(x)) \leftarrow C(x)\}),$
$b_2 = (f, \{B\}, \{C(x) \leftarrow B(f(x))\}), \qquad b_3 = (g, \{D\}, \emptyset).$

Note that b_1 is a root block, b_2, b_3 are child blocks; b_1 has 2 forward rules, b_2 has a backward rule, and that b_3 has no rules.

To ease the presentation, we assume that each function symbol f is indexed by $\mathsf{i}(f)$, that the function i is bijective, and w.l.o.g. that the set of indices of all functions in any set $\mathcal{B}$ of blocks is an initial segment of the positive integers.

Definition 4.2. (Block tree) Let $\mathcal{B}$ be a block set where k function symbols occur. Then a *$\mathcal{B}$-tree* is any k-ary $\mathcal{B}$-labeled tree $\mathcal{T} = (T, \mathcal{L})$ satisfying the following properness conditions: (i) $\mathcal{L}(\epsilon)$ is a root block, and (ii) for all $x{\cdot}c \in T$, $\mathcal{L}(x{\cdot}c) = (\alpha, \mathcal{D}, \mathcal{R})$ is a child block with $\mathsf{i}(\alpha) = c$.

Example 4.2. (Cont'd) Assume $\mathsf{i}(f) = 1$ and $\mathsf{i}(g) = 2$, and $\mathcal{B} = \{b_1, b_2, b_3, b_f, b_g\}$ with $b_f = (f, \emptyset, \emptyset)$ and $b_g = (g, \emptyset, \emptyset)$. Take a 2-ary tree T with labeling $\mathcal{L}(\epsilon) = b_1$, $\mathcal{L}(1) = b_2$, $\mathcal{L}(2) = b_3$, $\mathcal{L}(y) = b_f$ for all $y \in T$ with $y = x'{\cdot}1$ and $x' \neq \epsilon$, and $\mathcal{L}(y) = b_g$ for all $y \in T$ with $y = x'{\cdot}2$ and $x' \neq \epsilon$. Then $\mathcal{T} = (T, \mathcal{L})$ is a $\mathcal{B}$-tree. Another $\mathcal{B}$-tree $\mathcal{T}'$ can be obtained, e.g., by setting $\mathcal{L}(11) = b_2$ instead of $\mathcal{L}(11) = b_f$.

To obtain the encoded interpretation and program, we first translate nodes of trees to terms. To this end, let $x = k_1 \cdots k_n$ be a word over $\mathbb{N}$. Then $\mathsf{term}(x) = f_n(\ldots f_1(c) \ldots)$, where $\mathsf{i}(f_j) = k_j$ for every $j \in \{1, \ldots, n\}$ (note that $\mathsf{term}(\epsilon) = c$).

Definition 4.3. (Associated interpretation) For a $\mathcal{B}$-tree $\mathcal{T} = (T, \mathcal{L})$, we define *its associated interpretation* $\mathsf{int}(\mathcal{T})$ as

$$\mathsf{int}(\mathcal{T}) = \{A(\mathsf{term}(x)) \mid x \in T \wedge \mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R}) \wedge A \in \mathcal{D}\}.$$

Example 4.3. (Cont'd) Observe that $\mathsf{int}(\mathcal{T}) = \{A(c), C(c), B(f(c)), D(g(c))\}$ and $\mathsf{int}(\mathcal{T}') = \mathsf{int}(\mathcal{T}) \cup \{B(f(f(c)))\}$.

Definition 4.4. (Associated program) Let $r_{\downarrow t}$ denote the grounding a one-variable rule by a ground term t. Then for a $\mathcal{B}$-tree $\mathcal{T} = (T, \mathcal{L})$ *its associated program* $\mathsf{prog}(\mathcal{T})$ is the smallest program closed under the following rules:

a) if $x \in T$, $\mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R})$, $r \in \mathcal{R}$, and r is a local or a forward rule, then $r_{\downarrow \mathsf{term}(x)} \in \mathsf{prog}(\mathcal{T})$;

b) if $x \in T$, $\mathcal{L}(x) = (\alpha, \mathcal{D}, \mathcal{R})$, $r \in \mathcal{R}$, r is a backward rule, and $x = y{\cdot}c$ with $c \in \mathbb{N}$, then $r_{\downarrow \mathsf{term}(y)} \in \mathsf{prog}(\mathcal{T})$;

Example 4.4. (Cont'd) For the $\mathcal{B}$-tree $\mathcal{T}$, $\mathsf{prog}(\mathcal{T}) = \{A(c) \leftarrow;$ $B(f(c)) \leftarrow A(c);$ $D(g(c)) \leftarrow C(c);$ $C(c) \leftarrow B(f(c))\}$. Moreover, $\mathsf{prog}(\mathcal{T}') = \mathsf{prog}(\mathcal{T}) \cup \{C(f(c)) \leftarrow B(f(f(c)))\}$.

The minimality of $\mathcal{B}$-trees is defined in the natural way.

Definition 4.5. (Minimal $\mathcal{B}$-tree) We say a $\mathcal{B}$-tree $\mathcal{T}$ is *minimal*, if $\{\mathsf{int}(\mathcal{T})\} = MM(\mathsf{prog}(\mathcal{T}))$, i.e., $\mathsf{int}(\mathcal{T})$ is the least model of $\mathsf{prog}(\mathcal{T})$ (note that $\mathsf{prog}(\mathcal{T})$ is non-disjunctive).

Example 4.5. (Cont'd) Note that $\{\mathsf{int}(\mathcal{T})\} = MM(\mathsf{prog}(\mathcal{T}))$, and hence $\mathcal{T}$ is minimal. On the other hand, $\mathcal{T}'$ is not minimal because $\mathsf{int}(\mathcal{T}) \subset \mathsf{int}(\mathcal{T}')$ and $\mathsf{int}(\mathcal{T})$ is a model of $\mathsf{prog}(\mathcal{T}')$.

To characterize consistency of normal core programs P, we just have to properly define the set $\mathcal{B}$ of blocks.

Definition 4.6. A *block for a normal core program P* is any block $(\alpha, \mathcal{D}, \mathcal{R})$, where α is a constant or a function from P, $\mathcal{D}$ is a set of predicates of P, and $\mathcal{R}$ is a rule set consisting of:

a) All f-forward rules in P, for all functions f of P.

b) In case $\alpha = c$, the rule $A(x) \leftarrow$ for each fact $A(c) \leftarrow$ in P. Otherwise, if α is a function f, all f-backward rules of P.

c) (Reduct) For each local rule $r \in P$ such that $B \notin \mathcal{D}$ for all $B(x) \in \mathsf{body}^-(r)$, the rule $\mathsf{head}(r) \leftarrow \mathsf{body}^+(r)$.

By construction, for any $\mathcal{B}$-tree $\mathcal{T}$, where $\mathcal{B}$ is the set of all blocks for P, we have $\mathsf{prog}(\mathcal{T}) = P^{\mathsf{int}(\mathcal{T})}$. Hence, if $\mathcal{T}$ is minimal, then $\mathsf{int}(\mathcal{T}) \in SM(P)$. On the other hand, for any $I \in SM(P)$ we can build a minimal $\mathcal{B}$-tree $\mathcal{T}$ with $\mathsf{int}(\mathcal{T}) = I$.

Theorem 4.1. *If $\mathcal{B}$ is the set of all blocks for a normal core program P, then $SM(P) = \{\mathsf{int}(\mathcal{T}) \mid \mathcal{T}$ is a minimal $\mathcal{B}$-tree$\}$. Therefore, P is consistent iff there exists a minimal $\mathcal{B}$-tree.*

4.2 Generating Minimal Trees

In this section we assume an arbitrary set $\mathcal{B}$ of blocks, and construct an automaton $A^{\mathcal{B}} = \langle \Sigma, Q, \delta, q_0, F \rangle$, with $\Sigma = \mathcal{B}$, which accepts exactly the minimal $\mathcal{B}$-trees. To this end, let $\mathsf{rules}(\mathcal{B})$ and $\mathsf{preds}(\mathcal{B})$ denote the sets of all rules and predicate names occurring in $\mathcal{B}$, respectively. Furthermore, let k be the number of function symbols occurring in $\mathcal{B}$.

States Besides the initial state q_0, the set of states Q contains the following:

- "test" states q^c and q^j for global testing of consistency and justification in a candidate tree;
- for each $A \in \mathsf{preds}(\mathcal{B})$, the states $q_A^{\in}$, $q_A^{\notin}$ and q_A^j for testing containment of predicates in a block and their justification;
- for each $r \in \mathsf{rules}(\mathcal{B})$, the states $q_r^{\in}$ and q_r^c for testing rule containment in a block and rule satisfaction;
- the states q^p and $q_0^p, \ldots, q_k^p$ to ensure properness.

Transitions The transition function δ is defined as follows:

- (Initial state) For each $\sigma \in \Sigma$, there is a transition from q_0:

$$\delta(q_0, \sigma) = (0, q^c) \wedge (0, q^j) \wedge (0, q_0^p) \wedge \textstyle\bigwedge_{i=1}^{k} \big((i, q_i^p) \wedge (i, q^p)\big).$$

Intuitively, from the initial state the automaton switches to states for testing consistency and minimality, and also for testing if the symbol at the root is indeed a root block and all the successors are proper child blocks, which is realized as follows.

- (Properness) For each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and $1 \leq i \leq k$,[2]

$$\delta(q_0^p, \sigma) = [\alpha \text{ is a constant}],$$
$$\delta(q_i^p, \sigma) = [\alpha \text{ is a function with } \mathsf{i}(\alpha) = i],$$
$$\delta(q^p, \sigma) = \textstyle\bigwedge_{i=1}^{k} \big((i, q_i^p) \wedge (i, q^p)\big).$$

Intuitively, the automaton fails if in the state q_0^p (resp., q_i^p) it reads a block that is not a root block (resp., not a child block with function f s.t. $\mathsf{i}(f) = i$). The marking state q^p ensures recursively that the properness test is performed at each node of the tree.

- (Consistency) The test for consistency is defined in 3 steps. First, via the test state q^c, the rules that need be satisfied

[2] $[E]$ stands for **true**, if E evaluates to true, and else for **false**.

are selected, and the state q^c itself is propagated to the children. To this end, for every $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ we have:

$$\delta(q^c, \sigma) = \bigwedge_{r\in\mathcal{R}} (0, q^c_r) \wedge \bigwedge_{i=1}^{k} (i, q^c).$$

In the second step we move to satisfaction of the selected rules: for every $\sigma \in \Sigma$ and $r \in \mathsf{rules}(\mathcal{B})$, we have

$$\delta(q^c_r,\sigma) = \begin{cases} \bigvee_{i=1}^{n} (0,q^{\notin}_{B_i}) \vee (0,q^{\in}_{A}), & \text{if } r = A(x) \leftarrow \vec{B}_n(x),\\ (0,q^{\notin}_{B}) \vee (\mathsf{i}(f),q^{\in}_{A}), & \text{if } r = A(f(x)) \leftarrow B(x),\\ (0,q^{\notin}_{B}) \vee (-1,q^{\in}_{A}), & \text{if } r = A(x) \leftarrow B(f(x)),\end{cases}$$

where $\vec{B}_n(x) = \{B_1(x), \dots, B_n(x)\}$. Roughly, the rule r is satisfied, if either A is in the label of the node (resp., of some child or the parent), or some B_i (resp., B) is not.

Finally, the test for containment of labels is as follows: for each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and each $A \in \mathsf{preds}(P)$ we have:

$$\delta(q^{\in}_A, \sigma) = [A \in \mathcal{D}], \qquad \delta(q^{\notin}_A, \sigma) = [A \notin \mathcal{D}].$$

• (Minimality testing) We use the test state q^j to globally ensure justification of labels. For each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ:

$$\delta(q^j, \sigma) = \bigwedge_{A\in\mathcal{D}} (0, q^j_A) \wedge \bigwedge_{i=1}^{k} (i, q^j)$$

Intuitively, when in state q^j, the automaton simultaneously enters the states q^j_A in order to find justification for each predicate in $\mathcal{D}$, and also propagates q^j to all the children.

For the second step, let $M(A)$ denote the set of all tuples $\langle d, r, L\rangle$, where $d \in \{-1, 0, 1, \dots, k\}$, $r \in \mathsf{rules}(\mathcal{B})$ has the predicate A in its head, and $L \subseteq \mathsf{preds}(\mathcal{B})$ such that:

- for local r, $d = 0$ and $L = \{B \mid B(x) \in \mathsf{body}(r)\}$;

- for f-forward r, $d = -1$ and $L = \{B \mid B(x) \in \mathsf{body}(r)\}$;

- for f-backward r, $d = \mathsf{i}(f)$ and $L = \{B \mid B(f(x)) \in \mathsf{body}(r)\}$.

Intuitively, we collect in $M(A)$ the predicates and the direction that provide the justification. Now for each $A \in \mathsf{preds}(\mathcal{B})$ and each $\sigma \in \Sigma$, we have

$$\delta(q^j_A, \sigma) = \bigvee_{(d,r,L)\in M(A)} \big((d, q^{\in}_r) \wedge \bigwedge_{B\in L} (d, q^j_B)\big).$$

Intuitively, the automaton guesses the rule that will be used to find the justification. Finally, we need the transition for the rule-containment state $q^{\in}_r$: for each $\sigma = (\alpha, \mathcal{D}, \mathcal{R})$ in Σ and rule $r \in \mathsf{rules}(\mathcal{B})$, we have $\delta(q^{\in}_r, \sigma) = [r \in \mathcal{R}]$.

Acceptance Condition We define the parity condition $F : \emptyset \subseteq \{q^p, q^c, q^j\} \subseteq Q$. To understand this, observe that runs of $A^{\mathcal{B}}$ can have 4 types of infinite paths: (i) paths where exactly one of q^p, q^c, q^j occurs infinitely often, and (ii) paths where for some subset $\{A_1, \dots, A_n\} \subseteq \mathsf{preds}(\mathcal{B})$ only the justification states $q^j_{A_1}, \dots, q^j_{A_n}$ occur infinitely often. To ensure minimality, paths (ii) must be forbidden as they postpone justification indefinitely.

Theorem 4.2. *Given a set $\mathcal{B}$ of blocks, $A^{\mathcal{B}}$ accepts exactly the minimal $\mathcal{B}$-trees, i.e., $L(A^{\mathcal{B}})$ is the set of all minimal $\mathcal{B}$-trees.*

Note that the number of states in $A^{\mathcal{B}}$, where $\mathcal{B}$ is the set of all blocks of a normal core program P, is linear in $|P|$, and $\mathcal{B}$ is exponential in $|P|$. As testing non-emptiness of 2ATAs is feasible in time exponential in the number of states and polynomial in the size of the input alphabet [Vardi, 1998], by Theorem 4.2 and Proposition 3.1, we get the next completeness result; hardness can be shown by encoding an alternating Turing machine with polynomially bounded space.

Theorem 4.3. *Testing consistency of normal core programs and of normal* BD*-programs under bounded predicate arities is* EXPTIME*-complete.*

5 Consistency in Disjunctive BD-Programs

We analyze here the disjunctive case, and extend the method of the previous section to disjunctive core programs. To this end, we first characterize the minimal models of positive disjunctive ground programs in terms of *split programs*.

Definition 5.1. (Split) Let I be an interpretation for a positive (disjunctive) ground program P. A non-disjunctive positive program P' is called a *split of P w.r.t. I* if P' results from P by

(a) replacing each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| > 1$ with a rule $h \leftarrow \mathsf{body}(r)$, where $h \in \mathsf{head}(r) \cap I$ is picked arbitrarily, and

(b) replacing each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| = \emptyset$ with the constraint $\leftarrow \mathsf{body}(r)$.

By $SP(P, I)$ we denote the set of all splits of P w.r.t. I.

Intuitively, a split P' is obtained from P in two steps. First, we identify the disjunctive rules having several of head atoms true in I, and delete all but one such atom. Then the rules with no head atoms true in I are transformed into constraints. We can then characterize the minimal models of disjunctive ground programs as follows.

Theorem 5.1. *For any positive disjunctive ground program P it holds that $I \in MM(P)$ iff I is the least model of every $P' \in SP(P, I)$.*

Proof. For the "only if" case, observe that if $I \in MM(P)$, then I is also a model of every $P' \in SP(P, I)$, and that an arbitrary model of P' is also a model of P.

For the "if" case, suppose I is the least model of every $P' \in SP(P, I)$ and $I \notin MM(P)$. As already observed, I is a model of P. Hence, there must exist another model $J \subset I$ of P. Simply build a split P' of P w.r.t. I in two steps:

1. replace each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| > 1$ with a rule $h \leftarrow \mathsf{body}(r)$, where
   (i) $h \in \mathsf{head}(r) \cap I$, if $\mathsf{head}(r) \cap J = \emptyset$, and
   (ii) $h \in \mathsf{head}(r) \cap J$, if $\mathsf{head}(r) \cap J \neq \emptyset$;
2. replace each rule $r \in P$ such that $|\mathsf{head}(r) \cap I| = \emptyset$ with the constraint $\leftarrow \mathsf{body}(r)$.

As easily seen, J is a model of P', and thus I is not the least model of P'. Contradiction. ☐

Due to the theorem above, minimal model checking for positive disjunctive programs reduces to minimal model checking over a set of non-disjunctive programs. Building on this, we show decidability of BD-programs using trees whose nodes are labeled with *sets of blocks* (or, *hyperblocks*) instead of a single block. Intuitively, each *projection* of such a tree, obtained by arbitrarily discarding all but one block in each node, provides us with a $\mathcal{B}$-tree that encodes a different single split of a program. Consistency test for a program then amounts to finding a tree whose all projections are minimal $\mathcal{B}$-trees.

We first formally define the notion of projection over a tree.

Definition 5.2. (Projection) Let Σ be an alphabet, and let $\Sigma' \subseteq 2^{\Sigma}$. A tree $(T, \mathcal{L})$ over Σ is called a *Σ-projection of* a tree $(T, \mathcal{L}')$ over Σ', if for every node $n \in T$, $\mathcal{L}(n) \in \mathcal{L}'(n)$.

Trees having block trees as projections are defined next.

Definition 5.3. (Minimal hyperblock trees) A *hyperblock* is any set of blocks $(\alpha, \mathcal{D}, \mathcal{R})$ sharing the same α and $\mathcal{D}$.

Let $\mathcal{B}$ be a set of blocks with k function symbols occurring in it, and $\mathcal{H} \subseteq 2^{\mathcal{B}}$ be a set of hyperblocks. Then an *$\mathcal{H}$-tree* is any $\mathcal{H}$-labeled k-ary tree $\mathcal{T} = (T, \mathcal{L})$ satisfying the the following: (i) the blocks in $\mathcal{L}(\epsilon)$ are root blocks, and (ii) for all $x{\cdot}c \in T$, the blocks in $\mathcal{L}(x{\cdot}c)$ are child blocks $(\alpha, \mathcal{D}, \mathcal{R})$ with $\mathsf{i}(\alpha) = c$.

Note that $\mathcal{B}$-projections of $\mathcal{T}$ are (proper) $\mathcal{B}$-trees, and let $\mathsf{int}(\mathcal{T}) = \mathsf{int}(\mathcal{T}')$ for an arbitrary $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$ (note that for any other projection $\mathcal{T}''$, $\mathsf{int}(\mathcal{T}'') = \mathsf{int}(\mathcal{T}')$).

We say $\mathcal{T}$ is *minimal*, if each $\mathcal{B}$-projection of $\mathcal{H}$ is minimal.

To characterize stable models via hyperblock trees, we select suitable blocks and hyperblocks, taking into account splits.

Definition 5.4. A *block for a (disjunctive) core program P* is any block $(\alpha, \mathcal{D}, \mathcal{R})$, where α is the constant c or a function of P, $\mathcal{D}$ is a set of predicates of P, and $\mathcal{R}$ is a rule set consisting of:

a) All f-forward rules in P, for all functions f of P.

b) In case $\alpha = c$, the rule $A(x) \leftarrow$ for each fact $A(c) \leftarrow$ in P. Otherwise, if α is a function f, all f-backward rules of P.

c) (Reduct) Assuming $X = \{A(x) \mid A \in \mathcal{D}\}$, for each local rule $r \in P$ such that $\mathsf{body}^{-}(r) \cap X = \emptyset$:

   i. a rule $h \leftarrow \mathsf{body}^{+}(r)$ for an arbitrary $h \in \mathsf{head}(r) \cap X$, in case $\mathsf{head}(r) \cap X \neq \emptyset$, and

   ii. the constraint $\leftarrow \mathsf{body}^{+}(r)$, in case $\mathsf{head}(r) \cap X = \emptyset$.

A *hyperblock for P* is any $\subseteq$-maximal set of blocks $(\alpha, \mathcal{D}, \mathcal{R})$ for P sharing the same α and $\mathcal{D}$.

The following characterization is now an easy consequence of Theorem 5.1 and the above Definitions 5.3 and 5.4.

Theorem 5.2. *Let P be a core program, $\mathcal{B}$ the set of all blocks for P, and $\mathcal{H} \subseteq 2^{\mathcal{B}}$ the set of all hyperblocks for P. Then $SM(P) = \{\mathsf{int}(\mathcal{T}) \mid \mathcal{T}$ is a minimal $\mathcal{H}$-tree$\}$.*

Proof. (Sketch) Let $\mathcal{T}$ be a minimal $\mathcal{H}$-tree and $I = \mathsf{int}(\mathcal{T})$. Then I is the least model of $\mathsf{prog}(\mathcal{T}')$ for each $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$. Furthermore, due to the definition of hyperblocks for P, $\{\mathsf{prog}(\mathcal{T}') \mid \mathcal{T}'$ is a $\mathcal{B}$-projection of $\mathcal{T}\} = SP(P^{I}, I)$. Thus, I is the least model of every $P' \in SP(P^{I}, I)$, and, by Theorem 5.1, a minimal model of P^{I} and a stable model of P.

On the other hand, using Theorem 5.1, for any $I \in SM(P)$ we can easily build an $\mathcal{H}$-tree $\mathcal{T}$ with $\mathsf{int}(\mathcal{T}) = I$. ☐

To test existence of minimal $\mathcal{H}$-trees, and hence consistency in core programs, we resort to the automaton of the previous section, and to a projection-based notion of acceptance.

Definition 5.5. (Hyperacceptance) Let A be a 2ATA with an alphabet Σ, and let $\Sigma' \subseteq 2^{\Sigma}$. We say a tree $\mathcal{T}$ over Σ' *is Σ'-accepted by A*, if A accepts each Σ-projection of $\mathcal{T}$.

By combining Theorems 5.2 and 4.2, we get the following:

Proposition 5.3. *Let P be a core program, $\mathcal{B}$ the set of all blocks for P, and $\mathcal{H} \subseteq 2^{\mathcal{B}}$ the set of all hyperblocks for P. Then $SM(P)$=$\{\mathsf{int}(\mathcal{T}) \mid \mathcal{T}$ is $\mathcal{H}$-accepted by $A^{\mathcal{B}}\}$.*

Thus, consistency of P amounts to the existence of an $\mathcal{H}$-tree $\mathcal{T}$ such that $A^{\mathcal{B}}$ accepts each $\mathcal{B}$-projection of $\mathcal{T}$. To decide the latter, we apply well-known automata conversions:

- We first transform $A^{\mathcal{B}}$ into a 2ATA A_1 that accepts the complement of $L(A^{\mathcal{B}})$; this is well-known to be linear (the connectives in the transitions are inverted and the index of the sets in the parity condition increased by one).

- We transform A_1 into an equivalent *nondeterministic 1-way tree automaton (1NTA)* A_2 using the translation in [Vardi, 1998]; this causes an exponential blow-up in the number of states. In contrast to 2ATAs, the automaton A_2 moves only forward and its transitions are disjunctions of conjunctions.

- Building on A_2, we define a 1NTA A_3 that accepts exactly the trees $\mathcal{T}$ over $\mathcal{H}$ such that some $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$ is not accepted by $A^{\mathcal{B}}$. The components of A_3 are the ones of A_2 except the alphabet Σ_3, which is $\Sigma_3 = \mathcal{H}$, and the transition relation δ_3, which is $\delta_3(q, \sigma) = \bigvee_{\alpha \in \sigma} \delta_2(q, \alpha)$ for each state $q \in Q_3$ and symbol $\sigma \in \Sigma_3$. Intuitively, when scanning an $\mathcal{H}$-labeled tree $\mathcal{T}$, A_3 simulates a run of A_2 on some $\mathcal{B}$-projection $\mathcal{T}'$ of $\mathcal{T}$, and accepts $\mathcal{T}$ iff A_2 accepts some $\mathcal{T}'$.

- By complementing A_3, we obtain in linear time a 1ATA A_4 that accepts a tree $\mathcal{T}$ over $\mathcal{H}$ iff $\mathcal{T}$ is $\mathcal{H}$-accepted by $A^{\mathcal{B}}$.

The number of states and the alphabet $\mathcal{H}$ in the final automaton A_4 are exponential in $|P|$ (note that each set of predicate names from P together with a function or the constant c induces one hyperblock). Emptiness of a 1ATA is decidable in exponential time in the number of states and polynomial time in the size of the alphabet [Vardi, 1998]; overall, this yields double exponential time in $|P|$. By Proposition 5.3, we then obtain our main result on consistency testing; the hardness part can be proved by an encoding of an alternating Turing machine with exponentially bounded space.

Theorem 5.4. *Testing consistency of disjunctive core programs and of disjunctive* BD*-programs under bounded predicate arities is* 2EXPTIME*-complete.*

6 Further Results

Query entailment $P \models_{\mu} \exists \vec{x}.A(\vec{y})$ is efficiently reducible to (in)consistency testing in BD-programs, and thus has the same complexity. Briefly, if $A(\vec{y})$ is a y-atom and y is a variable, in cautious mode ($\mu = c$), we can use a constraint $\leftarrow A(\vec{y}), D(y)$, and in brave mode ($\mu = b$) rules $B(c) \leftarrow not\ B(c)$, $B(x) \leftarrow B(f(x))$, $B(y) \leftarrow A(\vec{y}), D(y)$, where D is an auxiliary domain predicate (cf. Example 3.3). The other cases can be easily reduced to this case. Note that, in fact, one can show 2EXPTIME-hardness of brave queries already over positive (disjunctive) core programs.

For BD-programs with unbounded predicate arities, we obtain an exponential increase in complexity (to completeness for 3EXPTIME, and 2EXPTIME in the non-disjunctive case). Intuitively, BD-programs are exponentially more succinct than core programs, and hence a reduction to a core program similar as in Proposition 3.1 is exponential in general.

However, the complexity of BD-programs is lower in case of a single function symbol (which still suffices to model a time line). Here, under bounded arities, consistency testing is EXPSPACE-complete in general, and PSPACE-complete in non-disjunctive case. Intuitively, the $\mathcal{B}$-trees degenerate to words over $\mathcal{B}$, and we can use *word* automata instead of tree automata, for which emptiness testing is accordingly easier.

7 Discussion and Conclusion

BD-programs enlarge the range of decidable ASP programs with function symbols, and constitute one of the few classes allowing for infinite stable models. Of the latter kind are *finitary* and *finitely recursive programs* [Bonatti, 2004], to which BD-programs are incomparable: they are neither finitary nor finitely recursive in general, but also do not capture any of the classes (as reasoning in them is semi-decidable in general).

Heymans [2006] used a similar automata construction as in Section 4 for *conceptual logic programs*, which have no function symbols and open answer set semantics. They also lack disjunction (in a usual sense) and are thus easier to handle.

Using our results, we can extend $\mathbb{FDNC}$ programs with *inverses* P^- of binary predicates P, i.e., $P^-(f(x), x)$ equals $P(x, f(x))$. In this way, some Description Logics with *inverse roles* (e.g., $\mathcal{ALCI}$) can be naturally captured and encoded efficiently into BD-programs. We just represent atoms $P(x, f(x))$ by $P_f(x)$ and $P^-(f(x), x)$ by $P^-_f(f(x))$, and use BD-rules $P_f(x) \leftarrow P^-_f(f(x))$ and $P^-_f(f(x)) \leftarrow P_f(x)$.

Our results may shed also new light on $Datalog_{nS}$, which extends $Datalog$ with n unary function symbols [Chomicki and Imielinski, 1993], and imply novel complexity results for this language. A detailed analysis remains for further work.

Other future work will also aim at more practical algorithms and applications (cf. Introduction and above).

References

[Baral, 2002] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2002.

[Baselice *et al.*, 2007] S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. In *ICLP-07*, LNCS 4670, pp. 89–103. Springer, 2007.

[Bonatti, 2004] P. A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.

[Calimeri *et al.*, 2008] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: Theory and implementation. In *ICLP-08*, LNCS 5366, pp. 407–424. Springer, 2008.

[Chomicki and Imielinski, 1993] J. Chomicki and T. Imielinski. Finite representation of infinite query answers. *ACM Trans. Database Syst.*, 18(2):181–223, 1993.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gen. Comput.*, 9(3/4):365–386, 1991.

[Heymans, 2006] Stijn Heymans. Decidable Open Answer Set Programming. Ph.D. Dissertation. Vrije Universiteit Brussel, 2006.

[Marek *et al.*, 1994] W. Marek, A. Nerode, and J. Remmel. The Stable Models of a Predicate Logic Program. *Journal of Logic Programming*, 21(3):129–153, 1994.

[Minker, 1988] J. Minker (ed). *Foundations of Deductive Databases & Logic Programming*. Morgan Kaufm., 1988.

[Simkus and Eiter, 2007] M. Simkus and T. Eiter. $\mathbb{FDNC}$: Decidable non-monotonic disjunctive logic programs with function symbols. In *LPAR-07*, LNCS 4790, pp. 514–530. Springer, 2007.

[Syrjänen, 2001] T. Syrjänen. Omega-restricted logic programs. In *LPNMR-01*, LNCS 2173, pp. 267–279. Springer, 2001.

[Thomas, 1990] W. Thomas. Automata on infinite objects. In *Handbook of Theor. Comp. Sci. (B)*, pp. 133–192. Elsevier, 1990.

[Vardi, 1998] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *Proc. ICALP-98*, LNCS 1443, pp. 628–641.

[Woltran, 2005] S. Woltran. Answer Set Programming: Model applications and proofs-of-Concept. Available at `http://www.kr.tuwien.ac.at/projects/WASP/report.html`.