Applicability of ASP-based Problem Solving on Tree Decompositions

Bernhard Bliem, Reinhard Pichler and Stefan Woltran
Institute of Information Systems, Vienna University of Technology
Favoritenstrasse 9–11; A-1040 Wien; Austria
{bliem, pichler, woltran}@dbai.tuwien.ac.at

Abstract

Many computationally hard problems in knowledge representation and reasoning (KRR) can become tractable if the graph structure underlying the problem instances at hand exhibits certain properties. An important structural parameter of this kind is treewidth, which measures the "tree-likeness" of a graph or, more generally, of a structure. By using a seminal result due to Courcelle, several fixed-parameter tractability (FPT) results in the area of AI and KRR have been proven in the last decade. To turn such theoretical tractability results into efficient computation in practice, suitable systems for conveniently implementing the necessary dynamic programming algorithms are required. A recent approach makes use of Answer Set Programming (ASP) for both the declarative description of dynamic programming algorithms and for solving the necessary subproblems. In this work, we prove that this new method can be used to efficiently solve any problem whose fixed-parameter tractability follows from Courcelle's Theorem.

1 Introduction

Intractability is a ubiquitous phenomenon in many areas of Computer Science – especially in Artificial Intelligence. Gottlob *et al.* [2010] have identified bounded treewidth as a key to the tractability of many reasoning problems including abduction and closed-world reasoning. More precisely, these problems can be characterized by a sentence in monadic second-order logic (MSO, for short), i.e., first-order predicate logic where in addition set variables are allowed. By Courcelle's Theorem [Courcelle, 1990] it follows that these problems are fixed-parameter linear (FPL, for short) w.r.t. the treewidth, i.e., they are solvable in linear time provided that the treewidth of the input structure is bounded by a constant.

In principle, there exist generic methods to automatically construct a concrete FPL algorithm for a problem from its MSO characterization. The classical approach works by exploiting the relationship between the model checking (MC) problem of MSO formulas on trees and finite tree automata. (see e.g. [Flum *et al.*, 2002; Klarlund *et al.*, 2002]). Recently an alternative game-theoretic approach (termed KLR approach below) has been presented by Kneis, Langer, and Rossmanith [2011] that underlies the system of Langer *et al.*. [2012]. One drawback of all these generic approaches is that they rely entirely on the MSO encoding of a problem; additional domain-specific knowledge that would improve performance is hard to incorporate. In contrast, several "tailor-made" algorithms have been developed specifically for particular problems. These algorithms usually employ dynamic programming over a *tree decomposition* of the given instance, propagating problem-specific data structures along this decomposition. Examples for such algorithms in the AI domain cover model counting [Samer and Szeider, 2010], belief revision [Pichler *et al.*, 2009], and argumentation [Dvorák *et al.*, 2012]. The disadvantage of this approach is its purely procedural nature, thus a practical implementation requires considerable programming effort. What is ultimately desired is a synthesis of the two paradigms, i.e. a *declarative framework* that allows the incorporation of *domain-specific knowledge* via dynamic programming.

In [Bliem *et al.*, 2012], we introduced the D-FLAT (*D*ynamic Programming *F*ramework with *L*ocal Execution of *A*SP on *T*ree Decompositions) approach that meets these requirements.[1] Its core idea is to use Answer Set Programming (ASP) [Gelfond and Leone, 2002; Marek and Truszczyński, 1999; Niemelä, 1999] to solve subproblems corresponding to subtrees of a tree decomposition and then to combine solutions. Our framework realizes this by heuristically generating a good tree decomposition of an input structure, executing a user-supplied ASP program at each tree decomposition node in a bottom-up traversal, and performing actions according to special predicates in the answer sets. The user of D-FLAT benefits from the convenient modeling language of ASP which leads to a relatively hassle-free specification of algorithms that exploit fixed-parameter tractability w.r.t. treewidth. Moreover, D-FLAT takes care of several technical tasks (decomposition of the input, propagating the data structures) and the underlying ASP engine ensures that even complex specifications are solved efficiently.

So far, it has remained unclear whether the D-FLAT approach is generally applicable to *every MSO-definable problem*. In this work, we show that the answer is positive. To

[1] D-FLAT is available as free software at `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`.

this end, we use the KLR approach to Courcelle's Theorem as theoretical underpinning for a D-FLAT encoding for MSO evaluation. This proves that *any problem whose fixed-parameter tractability is established via Courcelle's Theorem can be solved efficiently with ASP in the D-FLAT framework*.

The main challenges are the following: First, several modifications of the KLR approach will be necessary. Above all, that approach contains an isomorphism check between substructures induced by the domain elements present in some subtree of the tree decomposition in order to prune redundant branches of the tree representing the model checking game. This pruning is crucial to ensure the FPL behavior. We will show how this isomorphism check can be replaced by an equality check which is much better suitable for the ASP programs applied in the D-FLAT approach. As a byproduct, replacement of the isomorphism checks by equality checks yields a way to further enhance the KLR approach. Second, for the D-FLAT approach it is crucial to define which of the problem-solving steps are to be specified by the user (via an ASP encoding) and which general patterns of complex dynamic programming algorithms (as in the KLR approach) should be taken care of by the system. Here, the goal is to find a good balance between usability (specification of tailor-made algorithms should remain as simple as initially proposed in [Bliem *et al.*, 2012]) and expressiveness (the complex structures required for MSO evaluation should be expressible in D-FLAT encodings).

We can thus put our main contribution into a bigger picture: (i) the ASP-based D-FLAT approach provides an alternative method to fully capture an important class of fixed-parameter tractable problems (any property of finite structures expressible by an MSO sentence when applied to instances of bounded treewidth) allowing for a declarative, yet domain-specific, description of the actual FPL algorithm; (ii) due to the very nature of the D-FLAT approach, we demonstrate that the paradigm of ASP is not only a valuable tool for declaratively encoding a broad variety of problems (which are traditionally solved by a single call of an off-the-shelf ASP solver), but also for specifying *algorithms* that solve these problems on a decomposed structure (via multiple calls of an ASP solver) – an approach that should ultimately lead to significant performance gains on instances of small treewidth.

In this paper, we only give proof sketches. Supplementary material where proof details are worked out can be found at `http://dbai.tuwien.ac.at/proj/dynasp/dflat/supplementary-material-gkr13.pdf`.

2 Background

Answer Set Programming. ASP is a declarative language with roots in knowledge representation and reasoning (see, e.g., [Leone *et al.*, 2006]) where a *program* Π is a set of *rules* $a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$. The constituents of a rule $r \in \Pi$ are $h(r) = \{a_1, \ldots, a_k\}$, $b^+(r) = \{b_1, \ldots, b_m\}$ and $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. A set of atoms I satisfies a rule r iff $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a *model* of a set of rules iff it satisfies each rule. I is an *answer set* of a program Π iff it is a subset-minimal model of the program $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$ [Gelfond and Lifschitz, 1991].

```
{ in(X) : vertex(X) }.                          1
← edge(X,Y), in(X;Y).                           2
dominated(X) ← in(Y), edge(Y,X).                3
← vertex(X), not in(X), not dominated(X).       4
```

Listing 1: Computing independent dominating sets with ASP

With ASP programs, problem solving specifications following the *Guess & Check* principle can be represented succinctly. In particular, disjunctive rules (or rules that depend on each other) can be thought of as opening up the search space, whereas constraints (i.e., rules r with $h(r) = \emptyset$) impose restrictions that solutions must obey.

In this paper, we use the language of the grounder *Gringo* [Gebser *et al.*, 2010] where programs can contain variables that are instantiated by all ground terms (elements of the Herbrand universe, i.e., constants and compound terms containing function symbols) before a solver computes answer sets according to the propositional semantics stated above.

Example 1. *The program in Listing 1 solves the* INDEPENDENT DOMINATING SET *problem for directed graphs that are given as facts using the predicates* `vertex`*/1 and* `edge`*/2. Let* (V, E) *denote the input graph and recall that a set* $S \subseteq V$ *is an independent dominating set of* (V, E) *iff* $E \cap S^2 = \emptyset$ *and for each* $x \in V$ *either* $x \in S$ *or there is some* $y \in S$ *with* $(y, x) \in E$*. Note that this program not only solves the decision variant of the problem, which is NP-complete, but also allows for solution enumeration.*

Informally, the first rule (a so-called choice rule *having an empty body) states that* `in`*/1 is to be guessed to comprise any subset of* V*. The colon controls the instantiation of the variable* X *such that it is only instantiated with arguments of* `vertex`*/1 from the input. If for instance* $V = \{a, b, c\}$*, the grounder expands the rule to* `{ in(a), in(b), in(c) }` *leading to the intended guess of a subset of* V*. The rule in line 2 – where* `in(X;Y)` *is shorthand for* `in(X), in(Y)` *– checks the independence property. Lines 3 and 4 finally ensure that each vertex not in the guessed set is dominated by this set.*

Finite Structures and Tree Decompositions. A *finite structure* $\mathcal{A}$ over a set of relation symbols $\{R_1, \ldots, R_K\}$ is given by a finite domain $dom(\mathcal{A})$ and relations $R_i^{\mathcal{A}} \subseteq A^k$, where k denotes the arity of R_i. A *tree decomposition* of a structure $\mathcal{A}$ is a pair (T, χ) where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^{dom(\mathcal{A})}$ maps nodes to so-called *bags* such that *(1)* for every $a \in dom(\mathcal{A})$, there is an $n \in N$ with $a \in \chi(n)$, *(2)* for every relation symbol R_i and every tuple $(a_1, \ldots, a_k) \in R_i^{\mathcal{A}}$ there is an $n \in N$ with $\{a_1, \ldots, a_k\} \subseteq \chi(n)$, and *(3)* for every $a \in dom(\mathcal{A})$, the set $\{n \in N \mid a \in \chi(n)\}$ induces a connected subtree of T. The *width* of (T, χ) is defined as $\max_{n \in N} |\chi(n)| - 1$. The *treewidth* of $\mathcal{A}$ is the minimum width over all its tree decompositions.

For any fixed integer w, deciding if a structure has treewidth at most w and, if so, constructing a tree decomposition with width w can be done in linear time [Bodlaender, 1996], although these problems are intractable when w is part of the input [Arnborg *et al.*, 1987].

By slight abuse of notation, we write $n \in \mathcal{T}$ to denote that n is a node of $\mathcal{T}$. We write $\mathcal{T}_n$ and $\mathcal{A}_n$ to denote the subtree of $\mathcal{T}$ rooted at n and the substructure of $\mathcal{A}$ induced by the domain elements occurring in the bags in $\mathcal{T}_n$, respectively. We tacitly assume that each node $n \in \mathcal{T}$ is either a *leaf node*, an *introduce node* (having one child n' with $\chi(n') \subseteq \chi(n)$ and $|\chi(n) \setminus \chi(n')| = 1$), a *forget node* (having one child n' with $\chi(n') \supseteq \chi(n)$) and $|\chi(n') \setminus \chi(n)| = 1$) or a *join node* (having two children n_1, n_2 with $\chi(n) = \chi(n_1) = \chi(n_1)$), and that the root of $\mathcal{T}$ has an empty bag. By adding nodes, such a form can be obtained from any tree decomposition in linear time without increasing the width [Kloks, 1994].

Monadic Second-Order Logic. Monadic second-order logic (MSO) extends first-order logic by allowing quantification over set variables which are usually denoted by upper-case letters, whereas individual (i.e., first-order) variables are usually denoted by lower-case letters. Atomic formulas are of the form $X(x)$ or $R_i(x_1, \ldots, x_k)$ for a relation symbol R_i, set variable X and individual variables $x, x_1, \ldots, x_k$.

If a formula ϕ has free variables, we can evaluate it over a structure $\mathcal{A}$ given a variable assignment α that assigns all free individual variables to elements of $dom(\mathcal{A})$ and all free set variables to subsets of $dom(\mathcal{A})$. If ϕ is thus satisfied, we write $(\mathcal{A}, \alpha) \models \phi$. For sentences ϕ, we can abbreviate $(\mathcal{A}, \emptyset) \models \phi$ as $\mathcal{A} \models \phi$.

Courcelle's Theorem [Courcelle, 1990] states that for any fixed MSO sentence ϕ and integer k, given a structure $\mathcal{A}$ of treewidth at most k, one can decide $\mathcal{A} \models \phi$ in linear time.

Example 2. *Consider again the* INDEPENDENT DOMINATING SET *problem over structures with the binary relation* edge *and a domain consisting of vertices. It can be expressed with the MSO formula* $\exists S\big(\mathrm{ind}(S) \wedge \mathrm{dom}(S)\big)$*, where*

$$\mathrm{ind}(S) := \forall x \forall y \neg\big(S(x) \wedge S(y) \wedge \mathrm{edge}(x, y)\big)$$

and

$$\mathrm{dom}(S) := \forall x\Big(S(x) \vee \exists z\big(S(z) \wedge \mathrm{edge}(z, x)\big)\Big).$$

From this it follows that the problem parameterized by treewidth is fixed-parameter tractable. The main result of this paper shows that this fact can be exploited using D-FLAT.

3 Dynamic Programming for MSO

In this section, we give an overview of the game-theoretic proof of Courcelle's Theorem from [Kneis *et al.*, 2011] and introduce a significant modification suitable for our purposes.

3.1 MSO Model Checking Games

The starting point of the KLR approach is the naive evaluation of an MSO formula by inspecting all possible values of the quantified variables. For instance, a formula $\phi = \forall X \psi(X)$ is evaluated over a structure $\mathcal{A}$ by checking if $\psi(U)$ evaluates to true over $\mathcal{A}$ for all $U \subseteq dom(\mathcal{A})$. For quantifier rank q of ϕ and $|dom(\mathcal{A})| = d$, we thus get an upper bound $O((2^d + d)^q)$ on the number of cases that have to be considered.

The model checking (MC) game $\mathcal{G} = \mathcal{MC}(\mathcal{A}, \phi)$ for the MC problem $\mathcal{A} \models \phi$ of MSO is characterized by a rooted tree where each node (referred to as "position") corresponds to an MC problem:[2] The root of $\mathcal{G}$ corresponds to the MC problem $\mathcal{A} \models \phi$ itself. Now suppose that some node p in $\mathcal{G}$ corresponds to the MC problem $(\mathcal{A}, \alpha) \models \psi$. Then the child nodes of p correspond to all MC subproblems that have to be solved in order to decide $(\mathcal{A}, \alpha) \models \psi$. Thus, if $\psi = \forall X \psi'$ or $\psi = \exists X \psi'$, then p has a child node p_U for every $U \subseteq dom(\mathcal{A})$, and p_U corresponds to the MC problem $(\mathcal{A}, \alpha') \models \psi'$, where α' denotes the extension of α in which X is interpreted as U. Analogously, if $\psi = \forall x \psi'$ or $\psi = \exists x \psi'$, then p has one child node p_a for each $a \in dom(\mathcal{A})$, and p_a corresponds to the MC problem $(\mathcal{A}, \alpha') \models \psi'$, where α' extends α by interpreting x as a. If $\psi = \psi'_1 \wedge \psi'_2$ or $\psi = \psi'_1 \vee \psi'_2$, then p has two child nodes p_1 and p_2 corresponding to the MC problems $(\mathcal{A}, \alpha) \models \psi_1$ and $(\mathcal{A}, \alpha) \models \psi_2$, respectively. The leaf nodes in $\mathcal{G}$ correspond to the MC problem of single literals.

As each branch specifies an assignment over all variables occurring in ϕ, all truth values of the literals in the leaves are defined. We can then propagate the truth values bottom-up along the game tree in the obvious way to solve $\mathcal{A} \models \phi$.

Extended MSO Model Checking Game. Deciding $\mathcal{A} \models \phi$ via the MC game $\mathcal{MC}(\mathcal{A}, \phi)$ requires exponential time w.r.t. $|dom(\mathcal{A})|$. For structures of bounded treewidth one can do better. To this end, the KLR approach makes use of an extended model checking (EMC) game $\mathcal{EMC}(\mathcal{A}, \phi)$, where individual variables may remain uninterpreted.[3] The idea is that an EMC game is computed for each node of a tree decomposition $\mathcal{T}$ of $\mathcal{A}$ during a bottom-up traversal such that, given the EMC game at the root of $\mathcal{T}$, $\mathcal{A} \models \phi$ can be decided. The intuition of leaving a variable x uninterpreted at a node $n \in \mathcal{T}$ is that x will "later" (in the bottom-up traversal) be assigned a value from outside $\mathcal{A}_n$. Details of this bottom-up construction of EMC games are given in Section 3.3.

In principle, the evaluation of EMC games follows the same pattern as the evaluation of MC games, but the truth value of some MC subproblems can be undefined due to the fact that some individual variables have not been assigned a value. However, two important properties are shown in [Kneis *et al.*, 2011]: If the truth value of an EMC game $\mathcal{EMC}(\mathcal{A}, \phi)$ is defined, then this is indeed the correct value, i.e., it coincides with the truth value of $\mathcal{MC}(\mathcal{A}, \phi)$. Moreover, $\mathcal{EMC}(\mathcal{A}, \phi)$ can easily be converted into a corresponding MC game $\mathcal{MC}(\mathcal{A}, \phi)$ by deleting all branches from the tree of $\mathcal{EMC}(\mathcal{A}, \phi)$ that contain an uninterpreted variable.

3.2 Simplifications and Reductions

To guarantee an FPL upper bound on the time for MSO model checking over structures of bounded treewidth, the KLR approach defines the notion of "equivalent games" and deletes from each equivalence class all but one representative. More precisely, consider the EMC game $\mathcal{EMC}(\mathcal{A}_n, \phi)$ for some node $n \in \mathcal{T}$ and two sibling positions in this game, p_1 and

[2] MC problems can be viewed game-theoretically, where two players – the verifier and the falsifier – move between positions in the tree. We omit details here and refer to [Grädel, 2007].

[3] Actually, Kneis *et al.* [2011] additionally extended EMC games by specifying some set X of domain elements. However, for our purposes, the set X always denotes the bag $\chi(n)$ at a given node n of $\mathcal{T}$. We therefore ignore this set X here.

p_2, corresponding to the MC problem $(\mathcal{A}, \alpha_1) \models \psi$ and $(\mathcal{A}, \alpha_2) \models \psi$, respectively. Then p_1 and p_2 are "equivalent" (denoted as $p_1 \cong p_2$) if there exists an isomorphism h between $\mathcal{B}_1$ and $\mathcal{B}_2$ with $h(a) = a$ for all $a \in \chi(n)$, where $\mathcal{B}_i$ is the structure extending $\mathcal{A}$ by adding for each variable interpreted by α_i an accordingly interpreted constant or relation symbol to the signature. For two EMC games $\mathcal{G}_1$ and $\mathcal{G}_2$ with respective root nodes p_1 and p_2, we write $\mathcal{G}_1 \cong \mathcal{G}_2$ if $p_1 \cong p_2$ holds and there exists a bijection π from the subgames of $\mathcal{G}_1$ (i.e., the subtrees rooted at the children of position p_1) to the subgames of $\mathcal{G}_2$, s.t. $\mathcal{G}' \cong \pi(\mathcal{G}')$ for every subgame $\mathcal{G}'$ of $\mathcal{G}_1$.

Using this equivalence notion, the KLR approach defines a reduction procedure which recursively inspects any two siblings in an EMC game $\mathcal{G}$ and deletes one of them if they correspond to equivalent games. The desired FPL upper bound on the size of reduced EMC games is obtained in [Kneis *et al.*, 2011] by proving that only $f(\tau, \phi) \cdot ||\mathcal{A}||$ different equivalence classes of subgames of any position in $\mathcal{EMC}(\mathcal{A}, \phi)$ exist, where $f(\tau, \phi)$ is a function that depends on formula ϕ and treewidth τ of $\mathcal{A}$ (but not on the size of $\mathcal{A}$).

From Isomorphism to Equality. The isomorphism-based definition of equivalence turns out to be a severe obstacle for the D-FLAT approach since these isomorphism checks are tedious to realize in ASP. We therefore propose a different notion of "equivalence" of games. Our crucial observation is that for evaluating a formula ϕ it is irrelevant whether in some node corresponding to the MC problem $(\mathcal{A}_n, \alpha) \models \psi$ of an EMC game $\mathcal{EMC}(\mathcal{A}_n, \phi)$ an individual variable is interpreted as some element of $dom(\mathcal{A}_n) \setminus \chi(n)$ rather than another from this set, provided that both interpretations yield the same truth values for all subformulas that can so far be evaluated. In other words, if an individual variable is interpreted as some domain element that is only present in bags of nodes *below* the current node n during the bottom-up traversal, the precise value of that variable is irrelevant. All that is relevant is which subformulas this choice makes true or false. Similarly, once a domain element no longer appears in the current bag, it need no longer be explicitly stored in the interpretations of set variables.

We can justify this "forgetful" behavior by the fact that once individual variables have been assigned a domain element, this assignment is never changed. Hence, truth values of already evaluated subformulas stay the same. It is furthermore guaranteed that the truth values of subformulas that have not yet been evaluated can still be correctly determined. For instance, for an atom $R(x_1, \ldots, x_k)$ to have an undetermined truth value under α, at least one variable x_i must still be uninterpreted under α. Now if α interprets another variable x_j as some $a \in dom(\mathcal{A}_n) \setminus \chi(n)$, then that atom can never evaluate to true no matter which value from a bag further up in the tree decomposition is chosen for x_i. The reason for this lies in the definition of tree decompositions. Suppose the atom will eventually evaluate to true because there is an appropriate tuple containing a in the relation R. Then this tuple must jointly occur in some bag – but it cannot have been below or at n, as then the atom would not have an undetermined truth value. So it must occur in a bag further up in the tree decomposition. But this contradicts the requirement (3) of tree decompositions that all nodes whose bags contain a are connected.

To define an alternative equivalence criterion, we thus replace all elements from $dom(\mathcal{A}_n) \setminus \chi(n)$ with a new element "$*$". Moreover, in the interpretation of set variables, we only keep track of the domain elements contained in the current bag $\chi(n)$. Finally, we memorize the truth values of fully determined atoms in ϕ. Two EMC games are then equivalent if they are *equal*. It is easy to verify that our equivalence criterion allows for more reductions than the equivalence notion of [Kneis *et al.*, 2011]:

Theorem 1. *Let $\mathcal{T}$ be a tree decomposition and let n be a node in $\mathcal{T}$. Whenever two subtrees in the EMC game $\mathcal{EMC}(\mathcal{A}_n, \phi)$ are equivalent according to the isomorphism-based equivalence notion of [Kneis* et al.*, 2011], then they are also equivalent according to our equality-based equivalence criterion. The converse is, in general, not true.*

3.3 Bottom-up Computation of EMC Games

To evaluate $\mathcal{A} \models \phi$, we compute a reduced EMC game $\mathcal{G}_n$ for every node n in a tree decomposition $\mathcal{T}$ of $\mathcal{A}$ by means of a bottom-up traversal of $\mathcal{T}$. The actual evaluation of $\mathcal{A} \models \phi$ is done with the reduced EMC game $\mathcal{G}_r$, where r is the root of $\mathcal{G}$. In contrast to [Kneis *et al.*, 2011], we assume that MSO formulas are in prenex CNF. Hence, it suffices to keep track for every clause in the CNF if a literal that evaluates to true has already been found. We thus consider the clauses in the CNF as the leaf nodes of our EMC games. The most significant deviation from the algorithm of Kneis *et al.* [2011] is the different notion of equivalence as described in Section 3.2. The actual computation of the EMC game $\mathcal{EMC}(\mathcal{A}_n, \phi)$ will be detailed in Section 4 when we explain its ASP realization.

Theorem 2. *For the MC problem $\mathcal{A} \models \phi$, let $\mathcal{T}$ be a tree decomposition of $\mathcal{A}$. Moreover, let $\mathcal{EMC}(\mathcal{A}_n, \phi)$ denote the EMC game at a node n and let reduce$(\mathcal{EMC}(\mathcal{A}_n, \phi))$ denote the reduced EMC game obtained from $\mathcal{EMC}(\mathcal{A}_n, \phi)$ by replacing all elements of $\mathcal{A}_n \setminus \chi(n)$ by $*$, removing those elements from the interpretations of set variables, and then exhaustively deleting equal sibling subtrees.*

Then we can compute in time $O(f(\tau(\mathcal{T}), \phi))$ the reduced EMC game reduce$(\mathcal{EMC}(\mathcal{A}_n, \phi))$ from the reduced EMC game(s) at the child node(s) of n in $\mathcal{T}$. Here $\tau(\mathcal{T})$ denotes the width of $\mathcal{T}$ and f is a function not depending on $||\mathcal{A}||$.

Proof sketch. The upper bound on the complexity follows immediately from the application of the *reduce* operation at every node and the fact that only $O(f(\tau(\mathcal{T}), \phi))$ different reduced EMC games can exist. The correctness of the algorithm is proved in several steps: First, we need the correctness of the *reduce* operation, i.e., the result of an EMC game is not altered when deleting from any set of equivalent subtrees all but one element. We then need to prove for every node type of the node n in $\mathcal{T}$ that the computation of a reduced EMC game $reduce(\mathcal{EMC}(\mathcal{A}_n, \phi))$ from the reduced EMC game(s) at the child node(s) of n is correct. The proofs for the KLR algorithm (see Lemma 11 and 12 in [Kneis *et al.*, 2011]) can be easily carried over to our algorithm based on the altered equivalence criterion. Finally, we have to show that $\mathcal{A} \models \phi$ can indeed be decided by taking the reduced EMC game $\mathcal{G}_r$

at the root node r of $\mathcal{T}$, deleting all branches with an uninterpreted individual variable and evaluating the remaining MC game. Lemma 13 in [Kneis *et al.*, 2011], which proves the correctness of this step for the KLR algorithm, again can be carried over to our setting (in particular, the arguments in that proof remain valid if we use our equivalence criterion). □

4 MSO MC on Tree Decompositions with ASP

We now show that the D-FLAT approach can be applied to any MSO-definable problem. For this, we first explain how D-FLAT works – including natural extensions to the original framework introduced in [Bliem *et al.*, 2012] that make it more general. Then we present an encoding for solving MSO MC with it. Although this encoding could be seen as turning D-FLAT into a new generic MSO solver, its purpose is rather to prove the general applicability of D-FLAT.

Extending D-FLAT. In [Bliem *et al.*, 2012] we only considered problems in NP. Therefore we had to significantly extend D-FLAT to handle the complexity of MSO MC while keeping the basic methodology untouched. These extensions are, however, not ad-hoc modifications – D-FLAT as presented here is generic enough to accommodate all kinds of problems; in fact all examples from [Bliem *et al.*, 2012] carry over as special cases.

We equip each node n in a tree decomposition $\mathcal{T}$ of an input structure $\mathcal{A}$ with a so-called *i-tree*. By this we mean a tree where each node is associated with a set of ground terms called *items*. D-FLAT executes the user-supplied ASP program at each node $n \in \mathcal{T}$ (feeding it in particular the i-trees of the children of n as input) and parses the answer sets to construct the i-tree of n. This basic control flow is depicted in Figure 1. To keep track of its origin, each i-tree node m is additionally associated with a set of *extension pointers*, i.e., tuples referencing i-tree nodes from the child nodes of n that have given rise to m. For instance, if n has k children, the set of extension pointers of m consists of tuples $(p_1, \ldots, p_k)$, where each p_j is an i-tree node of the jth child of n. This allows us to obtain complete solutions at the end by combining the item sets along a chain of extension pointers.

As input to the encoding, D-FLAT declares the fact `final` if the current node $n \in \mathcal{T}$ is the root; `current`(v) for any $v \in \chi(n)$; if n has a child n', `introduced`(v) or `removed`(v) for any $v \in \chi(n) \setminus \chi(n')$ or $v \in \chi(n') \setminus \chi(n)$, respectively; `root`$(r)$ if n has a child whose i-tree is rooted at r; `sub`(m, m') for any pair of nodes m, m' in a child's i-tree, if m' is a child of m; and `childItem`(m, i) if the item set of node m from a child's i-tree contains the element i. Finally, D-FLAT also provides the input structure as a collection of ground facts.

The answer sets specify the i-tree of the current tree decomposition node. To be specific, each answer set describes an i-tree branch. Atoms of the following form are relevant for this: `length`(l) declares that the branch consists of $l+1$ nodes; `extend`(l, j) causes that j is added to the extension pointers of the node at depth l of the branch. `item`(l, i) states that the node at depth l of the branch contains i in its item set. All atoms using `extend`/2 and `item`/2 with the same depth argument constitute what we call a *node specification*.

To determine where branches diverge, D-FLAT uses the following recursive condition: Two node specifications coincide (i.e., describe the same i-tree node) iff *(1)* their depths, item sets and extension pointers are equal, and *(2)* both are at depth 0, or their parent node specifications coincide. In this way, an i-tree is obtained from the answer sets. It might however contain sibling subtrees that are equal w.r.t. item sets. If so, one of the subtrees is discarded and the extension pointers associated to its nodes are added to the extension pointers of the corresponding nodes in the remaining subtree. D-FLAT exhaustively performs this action to eliminate redundancies.

Example 3. *Listing 2 shows a D-FLAT encoding for* INDEPENDENT DOMINATING SET. *All i-trees have height 1 (due to line 1); their roots are always empty and their leaves contain items involving the function symbols* `in`*/1 and* `dominated`*/1. Suppose D-FLAT currently processes a forget node. Then there is one child i-tree. Let it consist of two branches whose respective leaf item sets are* $\emptyset$ *and* $\{$`in`(a), `dominated`$(b)\}$. *This i-tree is provided to the program in Listing 2 by means of the following input facts:*

```
root(r). sub(r,s1). sub(r,s2).
childItem(s2,in(a)).
childItem(s2,dominated(b)).
```

Each answer set of the program corresponds to a branch in the new i-tree, and each branch extends one branch from the child i-tree. The root of the new i-tree therefore always extends the root of the child i-tree (line 2). Which branch is extended is guessed in line 3. Lines 4 and 5 retain from the guess all items that apply to vertices still in the current bag. (All other items are simply forgotten, ensuring that the size of each i-tree is bounded by a function depending only on the decomposition width.) So if the branch with leaf "s2" is extended and vertex a *is forgotten, these lines cause that the answer set specifies the item* `dominated`(b), *but not* `in`(a).

Line 6 enforces the dominance condition. Note that it is not until a vertex is removed *that it can be established to violate this condition, since as long as a vertex is not removed potential neighbors dominating it could still be introduced. So, if instead a vertex* c *had been forgotten, the constraint in line 6 would eliminate the answer set extending branch "s2", since* c *was neither "in" nor "dominated".*

In introduce nodes, line 7 guesses whether the introduced vertex is "in" or "out" of the partial solution, and line 8 determines thereby dominated vertices. In line 9, the independence condition is enforced. Finally, line 10 ensures that in join nodes a pair of branches is only extended if these branches have not made conflicting choices ("in" or "out") for any of the common vertices.

When we originally introduced our approach in [Bliem *et al.*, 2012], we presented a special case of D-FLAT as it is discussed here. In fact, we used tables instead of i-trees and an answer set described a table row instead of an i-tree branch, as tables suffice to implement many dynamic programming algorithms on tree decompositions (cf., e.g., [Niedermeier, 2006]). D-FLAT as presented here is clearly more general, since tables can be seen as i-trees of height 1.

D-FLAT Encoding for MSO MC. We now show how EMC games are represented as i-trees, describe the representation

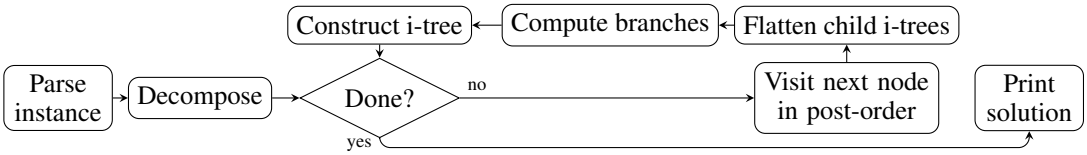

Figure 1: Control flow in D-FLAT

```
length(1).                                                                          1
extend(0,R) ← root(R).                                                              2
1 { extend(1,S) : sub(R,S) } 1 ← extend(0,R).                                       3
item(1,in(X)) ← extend(1,S), childItem(S,in(X)), current(X).                        4
item(1,dominated(X)) ← extend(1,S), childItem(S,dominated(X)), current(X).          5
← removed(X), extend(1,S), not childItem(S,in(X)), not childItem(S,dominated(X)).   6
{ item(1,in(X)) : introduced(X) }.                                                  7
item(1,dominated(X)) ← item(1,in(Y)), edge(Y,X), current(X).                        8
← edge(X,Y), item(1,in(X;Y)).                                                       9
← extend(1,S0;S1), childItem(S0,in(X)), not childItem(S1,in(X)).                    10
```
Listing 2: Computing independent dominating sets with D-FLAT

of MSO formulas and finally present our D-FLAT encoding for MSO MC. For the sake of readability, we only consider graphs as input structures using the predicates `vertex`/1 and `edge`/2. Our encoding can, however, be easily generalized.

We assume the formula to have quantifier blocks with $\exists$ at the outermost level. For an MSO MC problem $\mathcal{A} \models \phi$, each item set represents a position in $\mathcal{EMC}(\mathcal{A}, \phi)$. An item set at depth l of an i-tree encodes an assignment to the variables in the lth quantifier block. (Roots of i-trees remain empty.) Thus, we can associate a (partial) interpretation α_b of the matrix of ϕ with each branch b of an i-tree. Let n be the current node during a bottom-up traversal of a tree decomposition $\mathcal{T}$ of $\mathcal{A}$. α_b assigns $*$ to variables with values not in $\chi(n)$, but we can obtain all possible assignments α_b^+ without $*$ values by following the extension pointers.

We only use items of the following form: $\texttt{assign}(x, \_\texttt{nn})$ denotes that $\alpha_b(x) = *$; $\texttt{assign}(x, v)$ with $v \in \chi(n)$ denotes that $\alpha_b(x) = v$; $\texttt{assign}(X, v)$ denotes that $v \in \alpha_b(X)$; $\texttt{true}(c)$, which only occurs in leaf item sets, indicates that the clause c is true under α_b^+. For any individual variable x, the absence of any `assign`/2 item whose first argument is x means that x is still undefined.

MSO formulas are represented in ASP as follows. A fact of the form $\texttt{length}(i)$ declares that the number of alternating quantifier blocks is i. (This will cause each i-tree branch to have length i.) An individual variable x or set variable X that occurs in the ith quantifier block is declared by a fact of the form $\texttt{iVar}(i, x)$ or $\texttt{sVar}(i, X)$, respectively. The MSO atoms $x \in X$ and membership in the edge relation are represented in ASP as the terms $\texttt{in}(x, X)$ and $\texttt{edge}(x, y)$, respectively. Facts of the form $\texttt{pos}(c, a)$ or $\texttt{neg}(c, a)$ respectively denote that the atom a occurs positively or negatively in the clause c. For convenience, we have a fact $\texttt{clause}(c)$ for each clause c, and $\texttt{var}(i, x)$ for each individual or set variable x in the ith quantifier block.

Listing 3 shows our ASP encoding that is to be executed at each node $n \in \mathcal{T}$ to construct the i-tree of n representing $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_n, \phi))$. We now argue that this yields the correct result for each node type of n.

If n is a *leaf*, we guess a valid (partial) variable assignment without any $*$ values (lines 19 and 23) and declare the appropriate item sets (line 36). Additionally, we add the clauses that are satisfied by the assignment (cf. rules deriving `true`/1) into the leaf item set (line 37). Eventually, D-FLAT's processing of the resulting answer sets yields an i-tree representing the entire EMC game $\mathcal{EMC}(\mathcal{A}_n, \phi)$.

If n is an *introduce node* with child n', we guess a predecessor branch of the i-tree of n' (lines 14 and 15) whose assignment is preserved (line 20) and non-deterministically extended (lines 19 and 23). Satisfied clauses in the predecessor remain so (line 28). Again, clauses that become satisfied are determined and the appropriate item sets are filled.

If n is a *forget node*, we also guess a predecessor branch. We retain each `assign`/2 item unless it involves the removed vertex (line 20) and set the value of each individual variable that was assigned this vertex to $*$ (line 21). Determining satisfied clauses and declaring item sets proceed as before. This yields an i-tree where the removed vertex is removed from each set variable and individual variables previously set to that value are now assigned $*$.

If n is a *join node* with children n_1 and n_1, the bags $\chi(n) = \chi(n_1) = \chi(n_2)$ are identical. Here, we guess *a pair* of predecessor branches (lines 14 and 15). We generate $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_n, \phi))$ by combining "compatible" positions (p_1, p_2), where p_1 is a position in $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_{n_1}, \phi))$ and p_2 is a position in $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_{n_2}, \phi))$. We define "compatibility" of positions as follows. If p_1 and p_2 are the roots of the reduced EMC games, they are compatible. Now let p_1 and p_2 correspond to the MC problems $(\mathcal{A}, \alpha_1) \models \psi$ and $(\mathcal{A}, \alpha_2) \models \psi$, respectively, and suppose that the parents of p_1 and p_2 are compatible. First suppose that $\psi = \forall Y \psi'$ or $\psi = \exists Y \psi'$. Then p_1 and p_2 are compatible if α_1 and α_2 interpret Y identically, i.e., as the same subset of $\chi(n)$. For $\psi = \forall x \psi'$ or $\psi = \exists x \psi'$, the positions p_1 and p_2 are compatible if (1) x is uninterpreted in both α_1 and α_2, or (2) both interpret x as the same domain element from $\chi(n)$, or (3) one of α_1 and α_2 leaves x uninterpreted and the other interprets x as $*$. Line 25 enforces this compatibility. Note that p_1 and p_2

are incompatible if $\alpha_1(x) = \alpha_2(x) = *$ (line 24). This is because $*$ stands for a domain element in the substructure $\mathcal{A}_{n_1}$ or $\mathcal{A}_{n_2}$ that does not occur in the bag $\chi(n_1)$ or $\chi(n_2)$. By the definition of tree decompositions, the $*$ value of $\alpha_1(x)$ and the $*$ value of $\alpha_2(x)$ thus always stand for distinct elements. Since compatibility is enforced, the two preceding assignments can simply be unified to yield the assignment of the new branch (line 20). Now suppose that a branch π in $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_n, \phi))$ is obtained from branches π_1 and π_2 in $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_{n_1}, \phi))$ and $\mathit{reduce}(\mathcal{EMC}(\mathcal{A}_{n_2}, \phi))$, respectively. The set of clauses true along π is simply the union of the clauses true in π_1 and the clauses true in π_2 (line 28).

Finally, at the *root node* of $\mathcal{T}$ (which we assume to be a forget node with an empty bag, cf. Section 2), the child i-tree nodes are organized with `exists`/1, `forall`/1, `invalid`/1 and `bad`/1. The root is an "exists" node because ϕ is assumed to start with $\exists$. Along a branch, "exists" and "forall" nodes alternate because each non-root node covers all variables of a quantifier block. A node at depth l is "invalid" if it leaves some individual variable in the lth quantifier block uninterpreted, and it is "bad" if the subformula of ϕ starting with the lth quantifier block cannot be true. For this purpose, we start by labeling each non-invalid leaf with "bad" if it does not report all clauses to be satisfied (line 9). By following extension pointers, it can be verified that none of the interpretations represented by the respective branch satisfies the matrix of ϕ due to our bookkeeping of satisfied clauses. All leaves that are neither "invalid" nor "bad" conversely correspond to interpretations satisfying the matrix of ϕ. Using the alternation of "exists" and "forall" nodes, we then propagate truth values toward the root (lines 10–12): A "forall" node is "bad" iff one of its children is "bad", and an "exists" node is "bad" iff it has only "bad" or "invalid" children. Then it can be verified that $\mathcal{A} \models \phi$ holds iff the root of the child's i-tree is not "bad", as can be shown by induction. To ensure correctness and to only enumerate interpretations without undefined individual variables, the guessed predecessor branch must contain neither "bad" nor "invalid" nodes (lines 16 and 17). We say that D-FLAT accepts the input if the program executed at the root node has at least one answer set.

Theorem 3. *An MSO MC instance $\mathcal{A} \models \phi$ is positive exactly if D-FLAT, when executed on Listing 3 together with a declaration of ϕ, accepts input $\mathcal{A}$.*

Proof sketch. We can show by induction that the i-tree of any $n \in \mathcal{T}$ below the root of $\mathcal{T}$ can be used to construct $\mathcal{MC}(\mathcal{A}_n, \phi)$, and that the clauses satisfied by the interpretation corresponding to a branch of $\mathcal{MC}(\mathcal{A}_n, \phi)$ are exactly those in the respective leaf item set. If n is the child of the root node, we obtain $\mathcal{MC}(\mathcal{A}, \phi)$ in this way. If n is the root of $\mathcal{T}$, the propagation of truth values in the child i-tree (lines 1–12) can be shown to correspond to the propagation of truth values in $\mathcal{MC}(\mathcal{A}, \phi)$. If this propagation finally yields "false", line 16 ensures that no answer set exists because the child's i-tree root is then "bad". Otherwise, there is a branch in this i-tree consisting only of "good" nodes and D-FLAT accepts the input. □

```
assignedIn(X,S) ← childItem(S,assign(X,_)).                    1
% Evaluation (only in the root)                                2
itemSet(0,R) ← final, root(R).                                 3
itemSet(L+1,S) ← itemSet(L,R), sub(R,S).                       4
exists(S) ← final, root(S), sub(S,_).                          5
exists(S) ← forall(R), sub(R,S), sub(S,_).                     6
forall(S) ← exists(R), sub(R,S), sub(S,_).                     7
invalid(S) ← iVar(L,X), itemSet(L,S),                          8
  not assignedIn(X,S).
bad(S) ← length(L), itemSet(L,S),                              9
  not invalid(S), clause(C),
  not childItem(S,true(C)).
bad(S) ← forall(S), not invalid(S),                            10
  sub(S,T), bad(T).
bad(S) ← exists(S), not invalid(S),                            11
  not good(S).
good(S) ← exists(S), sub(S,T),                                 12
  not invalid(T), not bad(T).
% Guess a branch for each child node                           13
extend(0,R) ← root(R).                                         14
1 { extend(L+1,S) : sub(R,S) } 1 ←                             15
  extend(L,R), sub(R,_).
← extend(_,S), bad(S).                                         16
← extend(_,S), invalid(S).                                     17
% Preserve and extend assignment                               18
{ assign(X,V) : var(_,X) } ← introduced(V).                    19
assign(X,V) ← extend(_,S),                                     20
  childItem(S,assign(X,V)), not removed(V).
assign(X,_nn) ← extend(L,S),                                   21
  childItem(S,assign(X,V)), removed(V),
  iVar(L,X).
% Check that only compatible branches are joined and the       22
  resulting assignment is valid
← iVar(L,X), assign(X,V;W), V ≠ W.                             23
← extend(L,S0;S1), S0 ≠ S1,                                    24
  childItem(S0;S1,assign(X,_nn)).
← extend(L,S0;S1), var(L,X),                                   25
  childItem(S0,assign(X,V)),
  not childItem(S1,assign(X,V)), vertex(V).
% Determine clauses that have become true                      26
assigned(X) ← iVar(L,X), extend(L,S),                          27
  assignedIn(X,S).
true(C) ← extend(_,S), childItem(S,true(C)).                   28
true(C) ← pos(C,edge(X,Y)), assign(X,V),                       29
  assign(Y,W), edge(V,W).
true(C) ← neg(C,edge(X,Y)), assign(X,V),                       30
  assign(Y,W), vertex(V;W), not edge(V,W).
true(C) ← neg(C,edge(X,Y)), extend(_,S),                       31
  childItem(S,assign(X,V)), removed(V),
  not assigned(Y).
true(C) ← neg(C,edge(X,Y)), extend(_,S),                       32
  childItem(S,assign(Y,V)), removed(V),
  not assigned(X).
true(C) ← pos(C,in(X,Y)), assign(X,V),                         33
  assign(Y,V).
true(C) ← neg(C,in(X,Y)), assign(X,V),                         34
  vertex(V), not assign(Y,V).
% Declare resulting item sets                                  35
item(L,assign(X,V)) ← var(L,X), assign(X,V).                   36
item(L,true(C)) ← length(L), true(C).                          37
```

Listing 3: MSO model checking with D-FLAT

Given an input structure $\mathcal{A}$ whose treewidth is below some fixed integer, one can construct a tree decomposition of $\mathcal{A}$ in linear time. The total runtime for deciding $\mathcal{A} \models \phi$ for fixed ϕ is then linear, since the tree decomposition has linear size and the search space in each ASP call is bounded by a constant. Note that this, together with Theorem 2, amounts to an alternative proof of Courcelle's Theorem.

5 Conclusion

In this work, we have shown that the ASP-based approach of D-FLAT can be used to efficiently solve any problem whose fixed-parameter tractability follows from Courcelle's Theorem. To this end, we had to (i) adapt the recent game-theoretic proof of Courcelle's Theorem from [Kneis *et al.*, 2011], (ii) extend the D-FLAT system specification we gave in [Bliem *et al.*, 2012] and (iii) provide a suitable encoding for the MSO MC problem. All together, this shows that D-FLAT offers an alternative, purely declarative, way to develop dynamic programming algorithms on tree decompositions. Since the D-FLAT approach is centered around the ASP paradigm, our work provides a novel connection between ASP and fundamental methods from parameterized complexity with interesting research perspectives. One example is algorithm synthesis, where dynamic programming algorithms are obtained from a standard ASP problem description.

References

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, April 1987.

[Bliem *et al.*, 2012] Bernhard Bliem, Michael Morak, and Stefan Woltran. D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12(4-5):445–464, 2012.

[Bodlaender, 1996] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[Courcelle, 1990] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[Dvorák *et al.*, 2012] Wolfgang Dvorák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[Flum *et al.*, 2002] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[Gebser *et al.*, 2010] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Preliminary Draft. Available at `http://potassco.sourceforge.net`, 2010.

[Gelfond and Leone, 2002] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.

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

[Gottlob *et al.*, 2010] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, 174(1):105–132, 2010.

[Grädel, 2007] Erich Grädel. Finite model theory and descriptive complexity. In *Finite Model Theory and its Applications*, pages 125–230. Springer, 2007.

[Klarlund *et al.*, 2002] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.

[Kloks, 1994] Ton Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[Kneis *et al.*, 2011] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

[Langer *et al.*, 2012] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. Evaluation of an MSO-solver. In *Proc. ALENEX*, pages 55–63. SIAM / Omnipress, 2012.

[Leone *et al.*, 2006] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[Marek and Truszczyński, 1999] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[Pichler *et al.*, 2009] Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Belief revision with bounded treewidth. In *Proc. LPNMR 2009*, pages 250–263, 2009.

[Samer and Szeider, 2010] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.