Declarative Dynamic Programming as an Alternative Realization of Courcelle's Theorem

Bernhard Bliem, Reinhard Pichler, and Stefan Woltran

Institute of Information Systems, Vienna University of Technology
{bliem, pichler, woltran}@dbai.tuwien.ac.at

Abstract. Many computationally hard problems become tractable if the graph structure underlying the problem instance exhibits small treewidth. A recent approach to put this idea into practice is based on a declarative interface to specify dynamic programming over tree decompositions, delegating the computation to dedicated solvers. In this paper, we prove that this method can be applied to any problem whose fixed-parameter tractability follows from Courcelle's Theorem.

1 Introduction

Many computationally hard problems become tractable if the graph structure underlying the problem instance at hand exhibits certain properties. An important structural parameter of this kind is treewidth. By using a seminal result due to Courcelle [1] several fixed-parameter tractability results have been proven in the last decade. To turn such theoretical tractability results into efficient computation in practice, two contrary approaches can be found in the literature (see also the excellent upcoming survey [2]). Either the user designs a suitable dynamic programming algorithm that works directly on tree decompositions of the instances (see, e.g., [3]), or a declarative description of the problem in terms of monadic second-order logic (MSO) is used with generic methods that automatically employ a fixed-parameter tractable algorithm where the concepts of tree decomposition and dynamic programming are used "inside", i.e., hidden from the user (see, e.g., [4, 5] or the recent approach [6, 7]). The obvious disadvantage of the first strategy is its purely procedural nature, thus a practical implementation requires considerable programming effort. The second approach lacks possibilities to incorporate domain-specific knowledge which is typically exploited in tailor-made dynamic programming solutions and thus crucial for efficient solutions.

In order to combine the best of the two worlds, a recent LISP-based approach called `Autograph` (see, e.g., [8]) allows to specify the problem at hand via combinations of (pre-defined) fly-automata; hereby, domain-specific knowledge is incorporated on the automata level. Another recent approach employs Answer Set Programming (ASP) [9] in combination with a system called D-FLAT[1] [10]. In this approach, it is possible to entirely describe the dynamic programming algorithm by declarative means. D-FLAT heuristically generates a tree decomposition of an input structure and provides the data

[1] *D*ynamic Programming *F*ramework with *L*ocal Execution of *A*SP on *T*ree Decompositions. Available as free software at `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`.

structures that are propagated during dynamic programming. The task of solving each subproblem is delegated to an efficient ASP system that executes a problem-specific encoding. Such specifications typically reflect the problem solving intuition due to the possibility of using a *Guess & Check* technique, and the rich ASP language (including, e.g., aggregates) allows for concise, easy-to-read encodings.

So far, D-FLAT has only been applied to some sample problems lying in NP [10]. It has been left open if this approach is more generally applicable. In this work, we present a slight extension of the D-FLAT approach and prove that this new method can indeed be used to solve *any* MSO-definable problem parameterized by the treewidth in fixed-parameter linear time. We introduce semantic trees as a tool for MSO model checking (MC). Semantic trees are closely related to the approaches from [6, 11] but have properties that better suit our needs. Complementing the practically oriented exposition of D-FLAT in [10], the current work gives a theoretical result: We present an ASP-based description of a dynamic programming algorithm of the MSO MC problem via semantic trees and thus show the general applicability of the D-FLAT method.

2 Semantic Trees and Tree Decompositions

In this section we present our approach to MSO MC based on *semantic trees*, which are closely related to the game-theoretic techniques of [6] and the so-called *characteristic trees* of [11]. Below, we recall some basic notions and then highlight our method.

MSO model checking over finite structures. Let $\sigma = \{R_1, \ldots, R_K\}$ be a set of relation symbols. A *finite structure* $\mathcal{A}$ over σ (i.e., a "σ-structure", for short) is given by a finite domain $dom(\mathcal{A}) = A$ and relations $R_i^{\mathcal{A}} \subseteq A^\alpha$, where α denotes the arity of R_i.

We study the MSO model checking problem (i.e., the problem of evaluating an MSO sentence) over σ-structures. To simplify the presentation, we consider MSO sentences of the form $\phi = \exists Y_1 \exists z_1 \forall Y_2 \forall z_2 \ldots \exists Y_{n-1} \exists z_{n-1} \forall Y_n \forall z_n \psi$, s.t. n is even and ψ is a quantifier-free formula. Note that an *atom* in ϕ can either be of the form $R(z_{i_1}, \ldots, z_{i_\alpha})$ for some $R \in \sigma$ or of the form $Y_i(z_j)$. Let $At(\phi)$ denote the set of atoms occurring in ϕ.

An *interpretation* I of ψ over $\mathcal{A}$ is given by a tuple $(C_1, \ldots, C_n, d_1, \ldots, d_n)$, where $C_i \subseteq dom(\mathcal{A})$ is the interpretation of set-variable Y_i and $d_i \in dom(\mathcal{A})$ is the interpretation of the individual variable z_i. In a *partial interpretation*, we may assign the special value $undef$ to the individual variables z_i in ψ. The truth value $I(p)$ of an atom p in a partial interpretation I is defined in the obvious way: If at least one individual variable in the atom p is assigned the value $undef$ in I, then we also set $I(p) = undef$. Otherwise, $I(p)$ yields true or false exactly as for complete interpretations.

In order to systematically enumerate all possible interpretations for the quantifier-free part ψ of ϕ and to represent the truth value of ψ in each of these interpretations, we introduce the notion of *semantic trees*.

Definition 1. *For an MSO-formula ϕ and σ-structure $\mathcal{A}$, we define the* semantic tree *for ϕ and $\mathcal{A}$ as the following rooted, node-labeled tree with $2n + 2$ levels:*

- *Level 0 consists of the root.*
- *The nodes at levels 1 through $2n$ correspond to the variables $Y_1, z_1, Y_2, z_2, \ldots, Y_n$, z_n in this order.*

– *Each of the nodes at level $2n+1$ corresponds to the result of evaluating the atoms in ψ in one partial interpretation.*

The rank and label ℓ of each node N must satisfy the following conditions: the root has an empty label; every node at level $0, 2, 4, \ldots, 2n-2$ has $|2^{dom(\mathcal{A})}|$ child nodes, s.t. each subset $B \subseteq dom(\mathcal{A})$ occurs as the label of one of these child nodes; every node at level $1, 3, 5, \ldots, 2n-1$ has $|dom(\mathcal{A})|+1$ child nodes, s.t. each $d \in dom(\mathcal{A}) \cup \{undef\}$ occurs as the label of one of these child nodes; for every node N at level $2n$, we define $I(N)$ as the partial assignment where the labels along the path from the root to N are assigned to the variables $Y_1, z_1, Y_2, z_2, \ldots, Y_n, z_n$. Then every such node N has exactly one child node, whose label is a pair (At^+, At^-), s.t. At^+ and At^- are the sets of atoms in ψ that evaluate to true or, respectively, false in $I(N)$.

For an MSO-formula ϕ and σ-structure $\mathcal{A}$, we can use the corresponding semantic tree $\mathcal{S}$ to get a naive MSO MC procedure: first delete the subtree rooted at every node N from $\mathcal{S}$ whenever $\ell(N) = undef$; then reduce the MSO MC problem to a Boolean circuit evaluation problem by replacing the nodes in $\mathcal{S}$ by $\vee$ or $\wedge$ depending on whether the corresponding quantifier in the quantifier prefix of ϕ is existential or universal. The leaf nodes of the Boolean circuit are labeled "true" or "false" depending on the truth value of ψ in the interpretation represented by this branch.

Compression of semantic trees for a given tree decomposition. Of course, the MC procedure via semantic trees requires exponential time in the size of $\mathcal{A}$. We now show how semantic trees can be compressed in the presence of a tree decomposition of $\mathcal{A}$.

A *tree decomposition* of a structure $\mathcal{A}$ is a pair (T, χ) where $T = (V, E)$ is a (rooted) tree and $\chi : V \to 2^{dom(\mathcal{A})}$ maps nodes to so-called *bags* such that (1) for every $a \in dom(\mathcal{A})$, there is a $t \in V$ with $a \in \chi(t)$, (2) for every relation symbol R_i and every tuple $(a_1, \ldots, a_\alpha) \in R_i^{\mathcal{A}}$ there is a $t \in V$ with $\{a_1, \ldots, a_\alpha\} \subseteq \chi(t)$, and (3) for every $a \in dom(\mathcal{A})$, the set $\{t \in V \mid a \in \chi(t)\}$ induces a connected subtree of T. The latter is also known as the *connectedness condition*. The *width* of (T, χ) is defined as $\max_{t \in V}(|\chi(t)|) - 1$. The *treewidth* of $\mathcal{A}$ is the minimum width over all its tree decompositions. The notation $t \in \mathcal{T}$ expresses that t is a node of a tree decomposition $\mathcal{T}$. We write $\mathcal{T}_t$ and $\mathcal{A}_t$ to denote the subtree of $\mathcal{T}$ rooted at t, and the substructure of $\mathcal{A}$ induced by the domain elements occurring in the bags in $\mathcal{T}_t$, respectively.

By [12], we may assume that each node $t \in \mathcal{T}$ is of one of the following four types: It is either a *leaf node*, an *introduce node* (having one child t' with $\chi(t') \subseteq \chi(t)$ and $|\chi(t) \setminus \chi(t')| = 1$), a *forget node* (having one child t' with $\chi(t') \supseteq \chi(t)$ and $|\chi(t') \setminus \chi(t)| = 1$) or a *join node* (having two children t_1, t_2 with $\chi(t) = \chi(t_1) = \chi(t_2)$). Moreover, we may assume that the root of $\mathcal{T}$ has an empty bag.

The idea of our decision procedure for $\mathcal{A} \models \phi$ is to compute the semantic tree for every substructure $\mathcal{A}_t$ of $\mathcal{A}$. At the root node r of the tree decomposition, we thus get the semantic tree for the unrestricted structure $\mathcal{A}$, which we can then use for checking $\mathcal{A} \models \phi$ by a reduction to the Boolean circuit evaluation problem. We formally define this semantic tree for substructure $\mathcal{A}_t$ below.

Definition 2. *Consider an MSO-formula ϕ and σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$. For $t \in \mathcal{T}$, we say that $\mathcal{S}_t$ is the* local semantic tree at t *if $\mathcal{S}_t$ is the semantic tree of the MSO-formula ϕ and the induced substructure $\mathcal{A}_t$ of $\mathcal{A}$.*

To reach a fixed-parameter tractable algorithm w.r.t. treewidth, we introduce a compression of the semantic tree at each node t in the tree decomposition. The compression proceeds in two steps: First, we restrict the labels of the semantic trees to the domain elements present in $\chi(t)$. Second, if some node in a semantic tree has two child nodes with identical subtrees, then it suffices to retain only one of these subtrees.

Note that the concrete values of the labels at the internal nodes (i.e., the nodes corresponding to set variables or individual variables) in a semantic tree are irrelevant. Indeed, in the above reduction to Boolean circuits, only the tree structure of the semantic tree and the truth values (At^+, At^-) at the leaf nodes matter. As will be explained below, it is also convenient to slightly manipulate the truth values of some atoms which should be undefined according to the above definition of partial truth assignments. Since all subtrees with an undefined variable in one of the labels are ultimately removed from the semantic tree anyway, this has no effect on the evaluation of formula ϕ over structure $\mathcal{A}$. In summary, we get the following notion of *compressed, local semantic trees*.

Definition 3. *Consider an MSO-formula ϕ and σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$. For $t \in \mathcal{T}$, let $\mathcal{S}_t$ denote the local semantic tree at t. We call $\mathcal{C}_t$ a* compressed, local semantic tree *at t if $\mathcal{C}_t$ is obtained from $\mathcal{S}_t$ by applying rule L followed by rule A and then exhaustively applying rule R defined below:*

Rule L (changing **L**abels).

- *For every node corresponding to a set variable (i.e., levels $1, 3, \ldots, 2n-1$), the label $B \subseteq dom(\mathcal{A})$ is replaced $B \cap \chi(t)$.*
- *For every node corresponding to an individual variable (i.e., levels $2, 4, \ldots, 2n$), the label d is replaced by a special symbol $\star$ if $d \in dom(\mathcal{A}) \setminus \chi(t)$, and left unchanged otherwise, i.e. if $d \in \chi(t) \cup \{undef, \star\}$.*

Rule A (modification of **A**tom set At^-). *For every node at level $2n+1$, let I denote the interpretation along the path from the root to this node. In the label (At^+, At^-), replace At^- by $At^- \cup \{R(z_1, \ldots, z_\alpha) \in At(\phi) \mid \exists i, j$ s.t. $I(z_i) = undef$ and $I(z_j) = \star\}$.*

Rule R (eliminating **R**edundancy). *Let N be a node in $\mathcal{S}_t$ and let N_1, N_2 be two distinct child nodes of N. If the subtree rooted at N_1 and the subtree rooted at N_2 are identical, then we delete N_2 and the entire subtree rooted at N_2 from $\mathcal{S}_t$.*

The intuition of rule A is the following: Recall that the meaning of $I(z_j) = \star$ is that z_j is set to some value occurring in the subtree below node t in the tree decomposition but not in $\chi(t)$. The idea of letting $I(z_i) = undef$ is to set z_i to some value neither occurring $\chi(t)$ nor in the subtree below t. But then, by the connectedness condition of tree decompositions, we know that such atoms can never become true, no matter how the undefined variable will eventually be interpreted.

MSO model checking via compressed, local semantic trees. Given a finite structure $\mathcal{A}$ with a tree decomposition $\mathcal{T}$ and an MSO sentence ϕ, our MC procedure works in two steps: First, we compute a compressed, local semantic tree at every node t in $\mathcal{T}$ by a bottom-up traversal of $\mathcal{T}$. Then we evaluate ϕ over $\mathcal{A}$ by reducing the compressed, local semantic tree at the root node r of $\mathcal{T}$ to a Boolean circuit. Fixed-parameter linearity (w.r.t. the treewidth) of this algorithm is obtained as follows:

Theorem 1. *For the MSO model checking problem $\mathcal{A} \models \phi$, let $\mathcal{T}$ be a tree decomposition of $\mathcal{A}$. Then we can compute in time $O(f(\tau(\mathcal{T}), \phi) \cdot ||\mathcal{T}||)$ a compressed, local semantic tree $\mathcal{C}_t$ at every node t 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 computation of a compressed, local semantic tree $\mathcal{C}_t$ for every node $t \in \mathcal{T}$ proceeds in a bottom-up manner from the leaf nodes of $\mathcal{T}$ to the root. For this computation, we distinguish the four possible types that a node t of $\mathcal{T}$ can have:

(1) If t is a *leaf node*, it can be shown that $\mathcal{C}_t$ simply coincides with the local semantic tree $\mathcal{S}_t$ at t, i.e., none of the rules L, A, and R is applicable.

(2) Let t be an *introduce node* with child node t', s.t. $\chi(t') = \chi(t) \setminus \{b\}$. Then $\mathcal{C}_t$ is obtained from $\mathcal{C}_{t'}$ by copying subtrees of $\mathcal{C}_{t'}$ and modifying the labels of the copies as follows. Every node N in $\mathcal{C}_{t'}$ with $\ell(N) \subseteq \chi(t')$ gives rise to two nodes in $\mathcal{C}_t$: one with unchanged label $\ell(N)$ and one with label $\ell(N) \cup \{b\}$. Similarly, every node N in $\mathcal{C}_{t'}$ with $\ell(N) = \mathit{undef}$ gives rise to two nodes in $\mathcal{C}_t$: one with unchanged label undef and one with label b. Note that this corresponds to the intended meaning of the value undef, which is that a value shall be assigned to this individual variable "outside" the current subtree of the tree decomposition. For the adaptation of the truth values (At^+, At^-) at the leaf nodes of $\mathcal{C}_t$, the connectedness condition of tree decompositions is crucial. Finally, $\mathcal{C}_t$ is compressed via rule R.

(3) Let t be a *forget node* with child node t', s.t. $\chi(t) = \chi(t') \setminus \{b\}$. Then $\mathcal{C}_t$ is obtained from $\mathcal{C}_{t'}$ by first applying rule L. This means that we delete b from every set B in $\mathcal{C}_{t'}$. Moreover, if an individual variable is interpreted as b in $\mathcal{C}_{t'}$, we replace this interpretation by $\star$. For the truth values (At^+, At^-) at the leaf nodes of $\mathcal{C}_t$, it is now important to apply rule A from Definition 3. Finally, $\mathcal{C}_t$ is compressed via rule R.

(4) Finally, let t be a *join node* with child nodes t_1 and t_2. By definition of join nodes, we have $\chi(t) = \chi(t_1) = \chi(t_2)$. The nodes of $\mathcal{C}_t$ are obtained by combining "compatible" nodes of $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$. For an odd level $i < 2n$ in $\mathcal{C}_t$ (i.e., the labels of these nodes provide the interpretation of a set variable in ϕ), a node N_1 in $\mathcal{C}_{t_1}$ and a node N_2 in $\mathcal{C}_{t_2}$ are compatible if $\ell(N_1) = \ell(N_2)$. Compatibility in case of an even level $0 < i < 2n$ in $\mathcal{C}_t$ (i.e., a node whose label interprets an individual variable) holds if either (a) $\ell(N_1) = \ell(N_2)$ and $\ell(N_i) \neq \star$ or (b) one of $\ell(N_1), \ell(N_2)$ is undef. In case (a), the node N in $\mathcal{C}_t$ resulting from combining N_1 and N_2 simply gets the label $\ell(N) = \ell(N_1) = \ell(N_2)$. In case (b), the label of the resulting node N is set to $\ell(N_i)$ with $\ell(N_i) \neq \mathit{undef}$. Note that in (a), it is important to exclude the combination of nodes N_1 and N_2 with $\ell(N_1) = \ell(N_2) = \star$. This is due to the intended meaning of $\star$, which stands for some domain element in the subtree below t in the tree decomposition s.t. this value no longer occurs in the bag of t. Hence, the two occurrences of $\star$ in $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$ stand for different values. The label (At^+, At^-) at a leaf node of $\mathcal{C}_t$ is obtained from the labels of the corresponding nodes in $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$ by taking the component-wise union. Finally, we compress $\mathcal{C}_t$ via rule R. □

Our approach via (compressed) semantic trees has close links to the approaches based on extended MC games in [6] and on characteristic trees in [11]. The most significant difference is that we explicitly introduce a special symbol $\star$ for domain elements

not present anymore in a given bag of a tree decomposition. This allows us to define our reduction of semantic trees by a simple equality test, while the reduce-operation in [6] is based on an isomorphism criterion (which would not allow for a simple ASP realization). The characteristic trees in [11] are used in the context of structures of bounded rank-width and are computed by a bottom-up traversal of a given t-labeled parse tree decomposition. The reduction of characteristic trees is also based on an equality criterion. However, in contrast to tree decompositions, the notions of a "bag" and of a special symbol $\star$ (for domain elements not present anymore in some bag) are not applicable.

3 ASP and D-FLAT

In this section, we give brief introductions to Answer Set Programming (ASP) [9] and the D-FLAT system [10]. We thus set the stage for presenting our main result, i.e., that D-FLAT possesses enough expressive power for solving any MSO-definable problem parameterized by the treewidth in fixed-parameter linear time.

ASP is a declarative language where a *program* Π is a set of *rules*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \mathrm{not}\, b_{m+1}, \ldots, \mathrm{not}\, b_n.$$

The constituents of a rule $r \in \Pi$ are $h(r) = \{a_1, \ldots, a_k\}$, $b^+(r) = \{b_1, \ldots, b_m\}$ and $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Intuitively, r states that if an answer set contains all of $b^+(r)$ and none of $b^-(r)$, then it contains some element of $h(r)$. 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 $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$ [13].

ASP programs can be viewed as succinctly representing problem solving specifications following the *Guess & Check* principle. A "guess" can, for example, be performed using disjunctive rules which non-deterministically open up the search space. Constraints (i.e., rules r with $h(r) = \emptyset$), on the other hand, amount to a "check" by imposing restrictions that solutions must obey.

In this paper, we use the language of the grounder *Gringo* [14, 15] where programs may 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 following program solves the INDEPENDENT DOMINATING SET problem for graphs that are given as facts using the predicates `vertex` and `edge`.

```
{ 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
```

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.

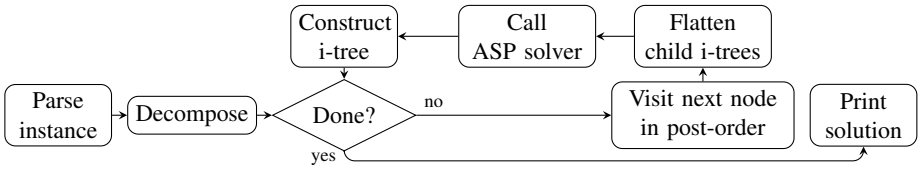

Fig. 1. Control flow in D-FLAT

Informally, the first rule (a so-called *choice rule* having an empty body) states that `in` 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` from the input. 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.

In order to take advantage of this Guess & Check approach in a decomposed setting, we make use of the D-FLAT system [10]. To perform dynamic programming on tree decompositions, D-FLAT needs data structures to propagate the partial solutions. To this end, it equips each node t 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 a user-supplied ASP program at each node $t \in \mathcal{T}$ (feeding it in particular the i-trees of children of t as input) and parses the answer sets to construct the i-tree of t. This procedure is depicted in Figure 1. To keep track of its origin, each i-tree node N is associated with a set of *extension pointers*, i.e., tuples referencing i-tree nodes from the child nodes of t that have given rise to N. For instance, if t has k children, the set of extension pointers of N consists of tuples $(N_1, \ldots, N_k)$, where each N_j is an i-tree node of the jth child of t. This allows us to obtain complete solutions by combining the item sets along a chain of extension pointers.

As input to the user's encoding, D-FLAT declares the fact `final` if the current node $t \in \mathcal{T}$ is the root; `current`(v) for any $v \in \chi(t)$; if t has a child t', `introduced`(v) or `removed`(v) for any $v \in \chi(t) \setminus \chi(t')$ or $v \in \chi(t') \setminus \chi(t)$, respectively; `root`$(r)$ if t has a child whose i-tree is rooted at r; `sub`(N, N') for any pair of nodes N, N' in a child's i-tree, if N' is a child of N; and `childItem`(N, i) if the item set of node N 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. Each answer set describes a branch in the i-tree. 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` and `item` 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 2. Listing 1.1 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` and `dominated`. Note that lines 7–10 resemble the program from Example 1, while the rest of the program is required for appropriately extending and combining partial solutions from child nodes.

Suppose D-FLAT is currently processing a forget node. Then there is one child i-tree. For illustration, assume it consists of two branches whose respective leaf item sets are $\emptyset$ and $\{\texttt{in}(a), \texttt{dominated}(b)\}$. This i-tree is provided to the program in Listing 1.1 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 5 and 6 derive which vertices are "in" or "dominated" according to this guess, and line 10 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 the vertex c has been removed, then the constraint in line 10 would eliminate the answer set extending branch "s2", since c is neither "in" nor "dominated". Lines 11 and 12 fill the leaf item set with only those items that apply to vertices still in the current bag. (This ensures that the maximum size of an i-tree only depends 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 $\texttt{dominated}(b)$, but not $\texttt{in}(a)$.

In introduce nodes, line 7 guesses whether the introduced vertex is "in" the partial solution or not. Line 8 enforces the independence condition and line 9 determines dominated vertices. Line 4 ensures that in join nodes a pair of branches is only extended if these branches agree on which of the common vertices are "in".

4 MSO MC on Tree Decompositions with ASP

We now present an encoding for MSO MC in the style of the approach from Section 2 in order to show that ASP with D-FLAT can solve any MSO-definable problem in linear time for bounded treewidth. In the following, let $\mathcal{A}$ and $\mathcal{T}$ denote the input structure and one of its tree decompositions, respectively. For the sake of readability, we only consider the case where $\mathcal{A}$ is a graph, given by the predicates `vertex` and `edge`. As in Section 2, we assume the MSO formula ϕ for which $\mathcal{A} \models \phi$ is to be decided to be of the form $\exists Y_1 \exists z_1 \forall Y_2 \forall z_2 \ldots \exists Y_{n-1} \exists z_{n-1} \forall Y_n \forall z_n \psi$. Here we additionally assume that ψ is in CNF. Our encoding can, however, be easily generalized. In particular, the quantifier alternation is not required in principle but facilitates presentation. Much could

```
length(1).                                                              1
extend(0,R) ← root(R).                                                  2
1 { extend(1,S) : sub(R,S) } 1 ← extend(0,R).                           3
← extend(1,S;T), childItem(S,in(X)), not childItem(T,in(X)).            4
in(X) ← extend(1,S), childItem(S,in(X)).                                5
dominated(X) ← extend(1,S), childItem(S,dominated(X)).                  6
{ in(X) : introduced(X) }.                                              7
← edge(X,Y), in(X;Y).                                                   8
dominated(X) ← in(Y), edge(Y,X).                                        9
← removed(X), not in(X), not dominated(X).                              10
item(1,in(X)) ← in(X), current(X).                                      11
item(1,dominated(X)) ← dominated(X), current(X).                        12
```

Listing 1.1. Computing independent dominating sets with D-FLAT

be done to improve the MSO model checker that emerges from this work; but this is outside the scope of this paper whose focus is on the general applicability of D-FLAT.

The formula ϕ is specified in ASP as follows. If the quantifier rank is i, then the fact $\texttt{length}(i{+}1)$ is declared. (This will cause each i-tree branch to have length $i{+}1$.) Each individual variable x or set variable X bound by the ith quantifier is declared by a fact of the form $\texttt{iVar}(i, x)$ or $\texttt{sVar}(i, X)$, respectively. The atoms $x \in X$ and membership in the edge relation are represented as $\texttt{in}(x, X)$ 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 supply a fact $\texttt{clause}(c)$ for each clause c, and $\texttt{var}(i, x)$ for each individual or set variable x bound by the ith quantifier.

Let t be the current node during a bottom-up traversal of a tree decomposition $\mathcal{T}$ of $\mathcal{A}$. The i-tree at t shall represent a compressed, local semantic tree. In particular, an item set of a (non-leaf) i-tree node shall encode the label of the respective semantic tree node. With each i-tree branch b we can thus associate a (partial) interpretation I_b of the variables in ϕ. I_b assigns $\star$ to variables with values not in $\chi(t)$, but we can extend it to all possible assignments I_b^+ without $\star$ values by following the extension pointers. As we assume ϕ to be in CNF, in the leaf of b we simply keep track of the clauses that have been satisfied by I_b^+ so far. We only use items of the following form: $\texttt{assign}(x, \texttt{\_nn})$ denotes that $I_b(x) = \star$; $\texttt{assign}(x, v)$ with $v \in \chi(t)$ denotes that $I_b(x) = v$; $\texttt{assign}(X, v)$ denotes that $v \in I_b(X)$; $\texttt{true}(c)$, which only occurs in leaf item sets, indicates that the clause c is true under I_b^+. For any individual variable x, the absence of any $\texttt{assign}$ item whose first argument is x means that x is still undefined.

Listing 1.2 shows the ASP encoding that is to be executed at each node $t \in \mathcal{T}$ to construct an i-tree representing $\mathcal{C}_t$, the compressed, local semantic tree at t. As input, the encoding is provided with a set of facts describing ϕ as well as $\mathcal{T}$ together with the i-trees from the children of t (see Section 3). We say that D-FLAT accepts the input $\mathcal{A}$ if the program executed at the root node of $\mathcal{T}$ has at least one answer set.

Theorem 2. *An MSO MC instance $\mathcal{A} \models \phi$ is positive iff D-FLAT, when executed on Listing 1.2 together with ϕ (represented in ASP as a set of facts), accepts input $\mathcal{A}$.*

Proof (Sketch). Let $\mathcal{A}$ be the input graph with a tree decomposition $\mathcal{T}$, let $t \in \mathcal{T}$ be the node currently processed by D-FLAT during the bottom-up traversal, and let $\mathcal{C}_t$ denote

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

Listing 1.2. MSO model checking with D-FLAT

the compressed, local semantic tree at t after executing the encoding at t. Again, $\mathcal{S}_t$ denotes the (non-compressed) local semantic tree at t, while $\mathcal{S}$ is the (complete) semantic tree for ϕ and $\mathcal{A}$. We first show that $\mathcal{C}_t$ is always constructed as desired according to the proof of Theorem 1. Then we show that from $\mathcal{C}_t$ we can always construct $\mathcal{S}_t$, and that this gives us $\mathcal{S}$ at the root of $\mathcal{T}$. The computation of $\mathcal{C}_t$ depends on the type of t.

(1) If t is a *leaf*, we guess a valid (partial) variable assignment without any $\star$ values (lines 18 and 22) and declare the appropriate item sets (line 35). Additionally, we add the clauses that are satisfied by the assignment (cf. rules deriving `true`) into the leaf item set (line 36). Eventually, D-FLAT's processing of the resulting answer sets (see Section 3) yields an i-tree representing $\mathcal{S}_t$, which coincides with $\mathcal{C}_t$.

(2) If t is an *introduce node* with child t', we guess a predecessor branch of the i-tree of t' (lines 13 and 14) whose assignment is preserved (line 19) and non-deterministically extended (lines 18 and 22). Already satisfied clauses remain so (line 27). Again, clauses that become satisfied are determined and the appropriate item sets are filled.

(3) If t is a *forget node*, we also guess a predecessor branch. We retain each `assign` item unless it involves the removed vertex (line 19), and we set the value of each individual variable that was assigned this vertex to $\star$ (line 20). Determining satisfied clauses and declaring item sets proceed as before. This yields an i-tree where the removed vertex is eliminated from the interpretation of each set variable, and individual variables previously set to that value are now assigned $\star$. Note that clauses might become satisfied due to the reasons for rule A from Section 2.

(4) If t is a *join node* with children t_1 and t_1, $\chi(t) = \chi(t_1) = \chi(t_2)$ holds. Here, we guess *a pair* of predecessor branches (lines 13 and 14). We generate $\mathcal{C}_t$ by combining "compatible" branches b_1 and b_2 from $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$, respectively. The notion of compatibility is the same as in the proof of Theorem 1, and enforced in lines 23 and 24. Thus the two assignments corresponding to b_1 and b_2 can simply be unified to yield the assignment of the new branch b (line 19). The set of clauses true under the assignment of b is now simply the union of the clauses true in b_1 and the clauses true in b_2 (line 27).

(5) If t is the *root node* of $\mathcal{T}$ (by assumption a forget node with an empty bag; see Section 2), the child i-tree nodes are organized with `exists`, `forall`, `invalid` and `bad`. Following the assumed form of the quantifier prefix of ϕ, non-leaf i-tree nodes at levels $4j$ and $4j+1$ (for $j \geq 0$) are marked with "exists", while those at levels $4j+2$ and $4j+3$ are marked with "forall". A non-leaf node at level l is "invalid" if the lth quantifier binds an individual variable left uninterpreted by that node, and it is "bad" if the subformula of ϕ starting after the lth quantifier cannot be true. For this purpose, we start by labeling each leaf with "bad" if it does not report all clauses to be satisfied (line 8). 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. We then propagate truth values toward the root (lines 9–11): 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. To ensure correctness and to only enumerate interpretations without undefined individual variables, the guessed predecessor branch must contain neither "bad" nor "invalid" nodes (lines 15 and 16).

Finally, we show that $\mathcal{A} \models \phi$ holds iff the root of the i-tree at the child of the root of $\mathcal{T}$ is not "bad". The i-tree of any $t \in \mathcal{T}$ below the root of $\mathcal{T}$ can be used to construct $\mathcal{S}_t$ by means of the extension pointers, as can be seen by induction. Furthermore, the clauses satisfied by the interpretation corresponding to a branch of $\mathcal{S}_t$ are exactly those in the respective leaf item set. If t is the child of the root node, we obtain $\mathcal{S}$ in this way. If t is the root of $\mathcal{T}$, the propagation of truth values in the child i-tree (lines 1–11) corresponds to the propagation of truth values in the Boolean circuit used for evaluation. If this propagation finally yields "false", line 15 ensures that no answer set exists because the i-tree root at the child of t is then "bad". Otherwise, there is a branch consisting only of nodes that are neither "bad" nor "invalid", and D-FLAT accepts the input. □

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 Theorems 1 and 2 together thus amount to an alternative proof of Courcelle's Theorem.

5 Conclusion

There is vivid interest in turning theoretical tractability results obtained via Courcelle's Theorem into concrete computation which is feasible in practice [2]. In this paper, we have shown that the ASP-based D-FLAT approach is one candidate for reaching this goal, having provided a realization of a suitable dynamic programming algorithm for the MSO model checking problem. Since MSO model checking is often impractical despite bounded treewidth [16], it is advisable to implement problem-specific algorithms. Experiments reported in [10] suggest that D-FLAT is a promising means to do so. In contrast to recent MSO-based systems [6, 7] where the problem is expressed in a monolithic way, D-FLAT allows to define the dynamic programming algorithm on a tree decomposition via ASP. Like in the Datalog approach [17], this admits a declarative specification while still being able to take advantage of domain knowledge. However, the approach in [17] aims at a single call to a Datalog engine, thus the very restrictive language of monadic Datalog is required to guarantee linear running times. Therefore, encoding the dynamic programming algorithm at hand is rather tedious (for instance, to handle set operations) making this approach less practical. In contrast, D-FLAT calls an ASP-solver in each node of the tree decomposition. This not only ensures the linear running times (assuming that D-FLAT encodings only use information from the current bag) but also allows one to take advantage of a richer modeling language, reducing the actual effort for the user. This leads to implementations of algorithms that leverage bounded treewidth in a natural way, as the examples in Section 3 and [10] show. In the current paper, we have shown that these were not just lucky coincidences – D-FLAT is indeed applicable to *any* MSO-definable problem. Future work in particular includes a comparison of the ASP-based D-FLAT approach with the LISP-based `Autograph` approach [8] regarding both the range of theoretical applicability and practical efficiency.

Acknowledgments. This work is supported by the Austrian Science Fund (FWF) projects P25518 and P25607. We also thank the anonymous referees for helpful comments.

References

1. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Inf. Comput. **85**(1) (1990) 12–75
2. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Practical algorithms for MSO model-checking on tree-decomposable graphs. Available at `http://tcs.rwth-aachen.de/~sikdar/index_files/article.pdf` (2013)
3. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press (2006)
4. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. J. ACM **49**(6) (2002) 716–752
5. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. **13**(4) (2002) 571–586
6. Kneis, J., Langer, A., Rossmanith, P.: Courcelle's theorem – a game-theoretic approach. Discrete Optimization **8**(4) (2011) 568–594
7. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: Proc. ALENEX, SIAM / Omnipress (2012) 55–63
8. Courcelle, B., Durand, I.: Computations by fly-automata beyond monadic second-order logic. CoRR **abs/1305.7120** (2013)
9. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12) (2011) 92–103
10. Bliem, B., Morak, M., Woltran, S.: D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. TPLP **12**(4-5) (2012) 445–464
11. Langer, A., Rossmanith, P., Sikdar, S.: Linear-time algorithms for graphs of bounded rankwidth: A fresh look using game theory - (extended abstract). In: Proc. TAMC. Volume 6648 of LNCS., Springer (2011) 505–516
12. Kloks, T.: Treewidth: Computations and Approximations. Volume 842 of LNCS. Springer (1994)
13. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4) (1991) 365–386
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. Preliminary Draft. Available at `http://potassco.sourceforge.net` (2010)
15. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2012)
16. Frick, M., Grohe, M.: The complexity of first-order and monadic second-order logic revisited. Ann. Pure Appl. Logic **130**(1-3) (2004) 3–31
17. Gottlob, G., Pichler, R., Wei, F.: Monadic datalog over finite structures of bounded treewidth. ACM Trans. Comput. Log. **12**(1) (2010)