Implementing Courcelle's Theorem in a Declarative Framework for Dynamic Programming

Bernhard Bliem*, Reinhard Pichler, and Stefan Woltran

Institute of Information Systems, TU Wien
{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 Answer Set Programming that allows us to specify dynamic programming over tree decompositions in this language, 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.

Keywords: Answer Set Programming, Tree Decomposition, Monadic Second-Order Logic, Courcelle's Theorem, Fixed-Parameter Tractability

1 Introduction

Many computationally hard problems become tractable if the graph structure underlying the problem instance at hand exhibits certain properties [13, 15, 31]. An important structural parameter of this kind is treewidth [33, 6, 7]. By using a seminal result due to Courcelle [11] several fixed-parameter tractability (FPT) 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 survey [27]). Either the user designs a suitable dynamic programming algorithm that works directly on tree decompositions of the instances (see, e.g., [31]), 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., [14, 22] or the recent approach [24, 26]). 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 to improve performance.

In order to combine the best of the two worlds, a recent approach employs Answer Set Programming (ASP) [8, 29, 19, 32, 30] in combination with a system called

* Corresponding author. Phone: +43-1-58801-740019. E-mail: bliem@dbai.tuwien.ac.at

D-FLAT[1] [2, 4]. 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 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.

Originally, D-FLAT has only been applied to some sample problems lying in NP [4]. The experiments conducted in that work showed that, when the treewidth of the instances is small, D-FLAT indeed outperforms state-of-the-art systems on several problems in NP. However, it has been left open if this approach is more generally applicable. In this work, we present an extended version 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 do so by showing how the MSO model checking (MC) problem can be implemented in D-FLAT by following the ideas presented in [24, 28]. We slightly modify the data structures used in these papers (*games* and *characteristic trees*) to obtain what we call *semantic trees*, which allow the concepts from [24, 28] to be implemented in D-FLAT with small modifications. Complementing the practically oriented exposition of D-FLAT in [2, 4], 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.

Technically, current ASP systems allow Turing machines to be simulated [18, 17], which seems to suggest that our expressibility result is a trivial consequence. However, Turing-completeness requires an unrestricted use of function symbols [9, 3], while we consider a fragment of ASP where function symbols of positive arity may not be nested. (In fact, the function symbols we employ serve only the purpose of convenient modeling, and we could easily adapt D-FLAT to eliminate function symbols at the expense of a couple of additional predicates.) Therefore our contribution is more than just an alternative implementation of the approach presented in [24]: We show in this work that D-FLAT allows tree-decomposition-based dynamic programming algorithms to be specified for all MSO-expressible problems, and to do so in a way that remains true to the guess-and-check methodology presented in [4] by not resorting to unrestricted use of function symbols. Due to this restriction on function symbols, the resulting programs stay largely solver-independent as they lie in the standard (decidable) ASP language. This is of particular interest because one of the aims of D-FLAT is to leave open the possibility of interchanging the internally used ASP system.

This work is structured as follows. In Section 2 we define semantic trees, which are the central concept in our approach to MSO MC, and show how they can be put to use in combination with tree decompositions. In Section 3 we give a brief introduction to Answer Set Programming and the D-FLAT system. Section 4 then shows how D-FLAT

[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/`.

can be used for implementing our MSO MC algorithm. We conclude the paper with a summary in Section 6.

The current paper is based on an extended abstract [5].

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 [24] and the so-called *characteristic trees* of [28]. We discuss the relationship of our method to these approaches in Section 5. Here, we first recall some basic notions and then highlight our method.

2.1 MSO Model Checking via Semantic Trees

Let $\sigma = \{R_1, \ldots, R_K\}$ be a set of relation symbols. A *finite structure* $\mathcal{A}$ over σ (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.

Many properties of finite structures can be expressed in MSO, which extends classical first-order logic by allowing quantification over sets of domain elements. In other words, besides individual variables (which we denote by lower-case letters) we may also use unary relational variables (which we denote by upper-case letters and also call *set variables*) for quantification in MSO. The semantics of MSO generalizes the first-order case in the obvious way. In particular, let S be a set variable, x be an individual variable and I be an interpretation that assigns a set of domain elements $I(S)$ to S and a single domain element $I(x)$ to x. Now the atom $S(x)$ evaluates to true under I if $I(x) \in I(S)$ holds. We refer to [15] for a precise formal definition of MSO.

Example 1. Given a graph $G = (V, E)$, the NP-complete INDEPENDENT DOMINATING SET problem asks if there is a set $S \subseteq V$ such that S is both an independent and a dominating set. Assuming a signature $\sigma = \{\texttt{edge}\}$, where $\texttt{edge}$ is the binary edge relation, this can be expressed by the following MSO formula.

$$\exists S \forall x \forall y \left(\neg(\texttt{edge}(x, y) \wedge S(x) \wedge S(y)) \wedge (\neg S(x) \rightarrow \exists z \left(S(z) \wedge \texttt{edge}(z, x)\right)\right)\right)$$

As we will show in this paper, the fact that this problem is expressible in MSO entails that the problem parameterized by the treewidth of the input graph can be solved using D-FLAT in fixed-parameter linear time.

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 ϕ in prenex normal form. The matrix of ϕ (i.e., the largest quantifier-free subformula) is referred to as ψ. We denote the set variables in ϕ by $Y_1, \ldots, Y_m$, and the individual variables by $z_1, \ldots, z_k$, such that the ith set or individual variable according to the quantifier prefix is called Y_i or z_i, respectively. 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 $Y_i(z_j)$. We denote the set of atoms occurring in ϕ by $At(\phi)$, and the number of quantifiers in ϕ by n.

An *interpretation* I of ψ over $\mathcal{A}$ is given by a tuple $(C_1, \ldots, C_m)$ along with a tuple $(d_1, \ldots, d_k)$, 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 $I(R(z_{i_1}, \ldots, z_{i_\alpha}))$ in a partial interpretation I is defined in the obvious way: If some z_{i_j} is assigned the value $undef$ in I, then $I(R(z_{i_1}, \ldots, z_{i_\alpha})) = undef$ holds. Otherwise, $I(R(z_{i_1}, \ldots, z_{i_\alpha}))$ yields true or false exactly as for complete interpretations. Likewise, $I(Y_i(z_j)) = undef$ holds if $I(z_j) = undef$, and otherwise the value of $I(Y_i(z_j))$ is the same 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 $n+2$ levels, where n is the number of quantifiers in ϕ. We say that each node at depth i with $1 \leq i \leq n$* corresponds *to the ith variable in the quantifier prefix of ϕ.*

The rank and label ℓ of each node N must satisfy the following conditions:

- *The root has an empty label.*
- *Every node whose child nodes correspond to a set variable has $|2^{dom(\mathcal{A})}|$ child nodes s.t. each subset of $dom(\mathcal{A})$ occurs as the label of one of these child nodes.*
- *Each node whose child nodes correspond to an individual variable has $|dom(\mathcal{A})|+1$ child nodes s.t. each element of $dom(\mathcal{A}) \cup \{undef\}$ occurs as the label of one of these child nodes.*
- *For every node N at level n, we define I_N as the partial assignment where the labels along the path from the root to N are assigned to the variables s.t., for $1 \leq i \leq n$, the ith variable is set to the label of the ith node of that branch. 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 any subtree of $\mathcal{S}$ rooted at a node N with $\ell(N) = undef$; then reduce the MSO MC problem to a Boolean circuit evaluation problem by replacing each node whose children correspond to an existentially quantified variable by $\vee$, each node whose children correspond to a universally quantified variable by $\wedge$, and replace the parents of the leaves by $\vee$.[2] The leaf nodes of the Boolean circuit are labeled $\top$ or $\bot$ depending on the truth value of ψ in the interpretation represented by this branch.

The correctness of this procedure follows from the game-theoretic notions in [24]: Semantic trees can be seen as a special case of *extended model checking games*. Our deletion of subtrees of $\mathcal{S}$ rooted at nodes labeled *undef* is analogous to procedure $convert$ from [24]. By Lemma 1 from [24], this results in the (non-extended) model checking game over ϕ and $\mathcal{A}$ (i.e., a game where no position corresponding to a nullary

[2] The parents of leaves always have exactly one child and could therefore equivalently be replaced by $\wedge$ instead of $\vee$.

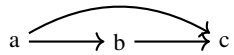

Fig. 1. A structure over the signature consisting of the binary edge predicate

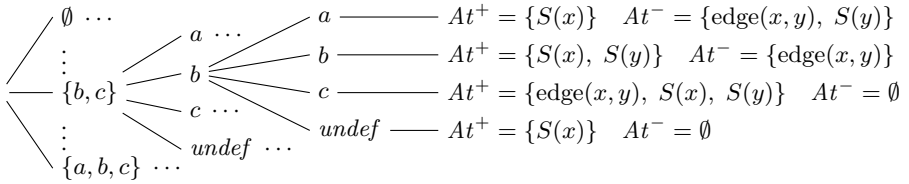

Fig. 2. Semantic tree for $\exists S \forall x \forall y \left(\mathrm{edge}(x, y) \rightarrow (S(x) \leftrightarrow \neg S(y)) \right)$ and the graph in Figure 1

symbol leaves that symbol uninterpreted). It is well known that this game evaluates to either $\top$ or $\bot$ (i.e., one of the players has a winning strategy) and that evaluating the game yields $\top$ if and only if $\mathcal{A} \models \phi$ (cf. [24]). This evaluation, corresponding to the procedure *eval* from [24], can easily be seen to be equivalent to the evaluation of Boolean circuits as described above.

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 this reduction to Boolean circuits, only the quantifier of each variable, the tree structure of the semantic tree and the truth values (At^+, At^-) at the leaf nodes matter.

Example 2. Consider the following MSO formula ϕ, which expresses the graph property of bipartiteness.

$$\exists S \forall x \forall y \left(\mathrm{edge}(x, y) \rightarrow (S(x) \leftrightarrow \neg S(y)) \right)$$

Let $\mathcal{A}$ be the structure depicted in Figure 1. The semantic tree for ϕ and $\mathcal{A}$ is shown in Figure 2. Some subtrees of this semantic tree have been omitted.

To solve the MSO MC problem by a reduction to Boolean circuit evaluation as described above, we first delete all subtrees rooted at a node labeled with "$undef$". Then we replace the root by $\vee$ (as the first variable is existentially quantified), the nodes at level 1 and 2 by $\wedge$ (as the second and third variable is universally quantified) and the nodes at level 3 by $\vee$. We replace the leaves by either $\top$ or $\bot$ depending on the truth value of $\mathrm{edge}(x, y) \rightarrow (S(x) \leftrightarrow \neg S(y))$ under the interpretation corresponding to the respective branch. The first two of the depicted leaves are thus replaced by $\top$, the third one is replaced by $\bot$ and the fourth one has been deleted as its parent node is labeled with "*undef*". Hence in the Boolean circuit the gate corresponding to the depicted node at depth 3 labeled "c" evaluates to $\bot$ and subsequently also its parent, which in turn makes the gate corresponding to the node labeled "$\{b, c\}$" evaluate to $\bot$. Indeed, every gate corresponding to a node at depth 1 evaluates to $\bot$, so the resulting Boolean circuit evaluates to $\bot$, which shows that $\mathcal{A} \not\models \phi$. This is the expected result since the graph from Figure 1 is not bipartite.

2.2 Compressing Semantic Trees using Tree Decompositions

Of course, the MC procedure via semantic trees requires exponential time in the size of $\mathcal{A}$. In the following, we introduce a few concepts that will enable us to achieve a fixed-parameter tractable (FPT) algorithm by compressing semantic trees 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 of $\mathcal{T}_t$, respectively. Furthermore, we write $\chi(\geq t)$ to denote the union of all bags $\chi(s)$ such that $s \in \mathcal{T}_t$. Moreover, let $\chi(> t)$ denote $\chi(\geq t) \setminus \chi(t)$.

By [23], 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 $\chi(t') \subset \chi(t)$ and $|\chi(t) \setminus \chi(t')| = 1$), a *forget node* (having one child t' with $\chi(t') \supset \chi(t)$ 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. We can transform an arbitrary tree decomposition into one having such a form in linear time.

We can obtain a decision procedure for $\mathcal{A} \models \phi$ by computing 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 now formally define this semantic tree for a substructure $\mathcal{A}_t$.

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 $\mathcal{A}$.*

Of course, using a tree decomposition to compute all local semantic trees is no better than the naive method for MSO MC via semantic trees, but it is the basis of our FPT algorithm for MSO MC. That is, we use it as a starting point for an algorithm that decides $\mathcal{A} \models \phi$ in time $O(f(\tau(\mathcal{T}), \phi) \cdot ||\mathcal{T}||)$, where $\mathcal{T}$ is a tree decomposition of $\mathcal{A}$, $\tau(\mathcal{T})$ denotes the width of $\mathcal{T}$ and f is a function not depending on $\mathcal{A}$. To this end, we introduce a compression of the local semantic tree at each node t in the tree decomposition. This compression will enable us to compute, at each tree decomposition node, an object whose size only depends on the formula and the input structure's treewidth, but not on the input structure's size.

The compression of the local semantic tree at a node $t \in \mathcal{T}$ proceeds in two steps. First, we restrict all labels specifying an interpretation of a variable to the domain elements present in $\chi(t)$:

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

Second, for any node that has two identical child subtrees, it suffices to retain only one of them: 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 $\mathcal{S}_t$. This compression allows us to achieve fixed-parameter tractability, and its correctness follows from Lemma 7 in [24].

To formalize the intuitions behind these steps for compressing local semantic trees, we need the following notion of an *s-tree*, which resembles a semantic tree but is less restrictive.

Definition 3. *For an MSO formula ϕ, a σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$ and a node $t \in \mathcal{T}$, we define an* s-tree at t *as a node-labeled tree with $n+2$ levels as follows. We say that each node at depth i with $1 \leq i \leq n$* corresponds *to the ith variable in the quantifier prefix of ϕ. The following conditions must be satisfied:*

- *The root has an empty label.*
- *Every node corresponding to a set variable has as label a subset of $\chi(t)$. Siblings do not necessarily have distinct labels.*
- *Every node corresponding to an individual variable has as label one of the elements of $dom(\mathcal{A}) \cup \{undef, \star\}$. The label $\star$ may be shared by several siblings.*
- *Every leaf node has as label a pair (At^+, At^-) with $At^+, At^- \subseteq At(\phi)$ and $At^+ \cap At^- = \emptyset$.*

In order to allow only such s-trees that relate to a local semantic tree in the way laid out by the compression steps above, we define the stricter notion of valid s-trees.

Definition 4. *Consider an MSO formula ϕ and σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$ and node $t \in \mathcal{T}$ along with the local semantic tree $\mathcal{S}_t$. We call an s-tree $\mathcal{C}$ a* valid s-tree *if no node in $\mathcal{C}$ has identical child subtrees and there exists a mapping μ from the nodes of $\mathcal{S}_t$ to the nodes of $\mathcal{C}$ with the following properties:*

1. *μ maps every node in $\mathcal{S}_t$ to a node in $\mathcal{C}$ at the same level.*
2. *Suppose that some inner node N in $\mathcal{S}_t$ is mapped to the node $\mu(N) = N'$ in $\mathcal{C}$. Moreover, let $\{N_1, \ldots, N_\alpha\}$ denote the set of child nodes of N in $\mathcal{S}_t$ and let $\{N'_1, \ldots, N'_\beta\}$ denote the set of child nodes of N' in $\mathcal{C}$. Then μ maps every N_i to some N'_j and every N'_j indeed occurs as the function value $\mu(N_i)$ of at least one N_i.*
3. *The following relationships between the label of N and of $\mu(N)$ hold:*
   - *If N corresponds to a set variable, then $\ell(\mu(N)) = \ell(N) \cap \chi(t)$.*

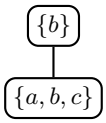

Fig. 3. A tree decomposition of the structure from Figure 1. (For instructional purposes explained in Example 3, the decomposition contains redundancies.)

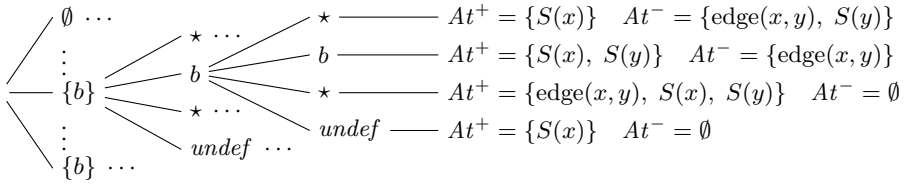

Fig. 4. A valid s-tree for the formula $\exists S \forall x \forall y \left(\mathrm{edge}(x, y) \rightarrow (S(x) \leftrightarrow \neg S(y)) \right)$ at the root of the decomposition from Figure 3

- *If N corresponds to an individual variable, then $\ell(\mu(N)) = \ell(N)$ holds if $\ell(\mu(N)) \in \chi(t) \cup \{undef\}$, and $\ell(\mu(N)) = \star$ otherwise.*
- *If N corresponds to a pair of atom sets, then $\ell(\mu(N)) = \ell(N)$.*

Example 3. Figure 3 shows a tree decomposition of the structure from Figure 1.[3] Figure 4 depicts a valid s-tree at the root of this decomposition for the MSO formula $\exists S \forall x \forall y \left(\mathrm{edge}(x, y) \rightarrow (S(x) \leftrightarrow \neg S(y)) \right)$. Although the labels $\star$, $\emptyset$ and $\{b\}$ occur in multiple sibling nodes, the s-tree does not contain redundancies in the form of identical sibling subtrees. We can construct the required function μ by mapping any node from the semantic tree shown in Figure 2 (which is also the local semantic tree at the root of the decomposition) to that node in Figure 4 that is at the same position in the illustration.

Note that in this particular example the valid s-tree has no advantages to the respective local semantic tree in terms of size. In general, however, valid s-trees are significantly smaller than local semantic trees because (for a fixed formula) the size of the former is bounded by the treewidth, whereas the size of the latter depends on the size of the input structure. We will prove this fact in Theorem 1.

2.3 Computating Valid s-trees at Tree Decomposition Nodes

We now describe how to compute a valid s-tree at all nodes $t \in \mathcal{T}$ by a bottom-up traversal of a given tree decomposition $\mathcal{T}$. To this end, we treat the four node types of a tree decomposition separately (i.e., leaf nodes, forget nodes, introduce nodes and join nodes). For each of these types, we show how a valid s-tree can be constructed at

[3] Strictly speaking, the root of this decomposition is redundant as its bag is a subset of its child's bag. The depicted tree decomposition is, however, legal and serves to illustrate the fact that $\star$ may stand for different forgotten vertices.

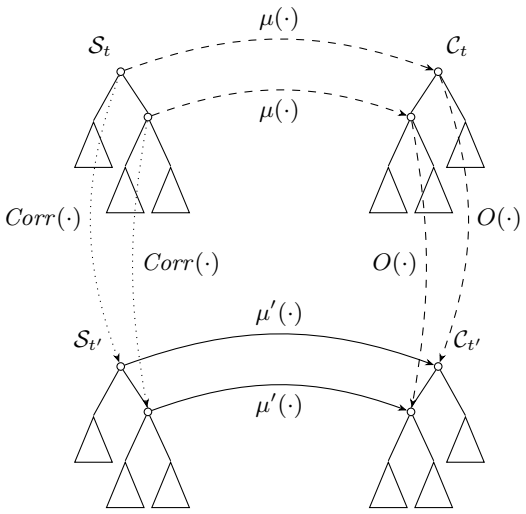

Fig. 5. Proof strategy for the lemmas about constructing a valid s-tree $\mathcal{C}_t$ at a tree decomposition node t with child t'

a decomposition node of the respective type, given a valid s-tree for each of the child nodes. We do this by constructing an s-tree using the valid s-trees from the children and then providing a mapping from the current decomposition node's local semantic tree to the constructed s-tree, as required by Definition 4. While the ideas underlying our algorithms have already been introduced in [24], our contribution is to show how they can be implemented using s-trees. In contrast to the more general *games* in [24], s-trees lend themselves to be used in conjunction with D-FLAT. This will allow us to solve MSO MC via D-FLAT, as we will show in Section 4.

This proof strategy is illustrated in Figure 5 for a node t having a sole child t' with a valid s-tree $\mathcal{C}_{t'}$. (The idea is the same for leaf or join nodes.) From $\mathcal{C}_{t'}$ we construct $\mathcal{C}_t$ depending on the type of t, as described in the respective lemma. Since $\mathcal{C}_{t'}$ is valid, we may assume that a mapping μ' from the nodes of the local semantic tree $\mathcal{S}_{t'}$ to $\mathcal{C}_{t'}$ exists. We use this fact for constructing a mapping μ from the nodes of the local semantic tree $\mathcal{S}_t$ to $\mathcal{C}_t$ in order to show that $\mathcal{C}_t$ is valid as well. In the lemma for introduce nodes (and similarly for join nodes), we make use of functions $O(\cdot)$ and $Corr(\cdot)$, which are also depicted in the illustration, to indicate the "origins" of nodes in $\mathcal{C}_t$ and $\mathcal{S}_t$, respectively.

Lemma 1. *Consider an MSO formula ϕ, a σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$, and a leaf node $t \in \mathcal{T}$. Then the valid s-tree at t coincides with the local semantic tree $\mathcal{S}_t$ at t.*

Proof. For leaf nodes t of the tree decomposition $\mathcal{T}$, we have $\chi(t) = \chi(\geq t)$. It is easy to see that $\mathcal{S}_t$ is also a valid s-tree: As the mapping μ required in Definition 4 we can choose the identity function. Moreover, no node in $\mathcal{S}_t$ has identical child subtrees since no two semantic tree nodes with common parent have identical labels.

Lemma 2. *Consider an MSO formula ϕ, a σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$, and a forget node $t \in \mathcal{T}$ whose child node is t'. Suppose that $\chi(t') \setminus \chi(t) = \{b\}$. Then a valid s-tree $\mathcal{C}_t$ at t is obtained from a valid s-tree $\mathcal{C}_{t'}$ at t' as follows:*

1. *For every node corresponding to a set variable, the label $B \subseteq dom(\mathcal{A})$ is replaced by the label $B \setminus \{b\}$.*
2. *For every node corresponding to an individual variable, the label $d \in dom(\mathcal{A})$ is left unchanged for $d \neq b$ and replaced by $\star$ if $d = b$.*
3. *Finally, we delete any subtree having an identical sibling subtree until all such redundancies are eliminated.*

Proof. Let $\mathcal{S}_t$ and $\mathcal{S}_{t'}$ denote the local semantic trees at t and t', respectively. For a forget node t, we have $\chi(\geq t) = \chi(\geq t')$ and, therefore, $\mathcal{S}_t = \mathcal{S}_{t'}$. Since $\mathcal{C}_{t'}$ is a valid s-tree at t', there exists a mapping $\mu' \colon \mathcal{N}(\mathcal{S}_{t'}) \to \mathcal{N}(\mathcal{C}_{t'})$ with the properties specified in Definition 4. Now consider the s-tree $\mathcal{C}_t^*$ at t, which is obtained from $\mathcal{C}_{t'}$ by the steps 1 and 2 of this lemma. Clearly, these steps do not change the tree structure. In total, we have $\mathcal{N}(\mathcal{S}_t) = \mathcal{N}(\mathcal{S}_{t'})$ and $\mathcal{N}(\mathcal{C}_t^*) = \mathcal{N}(\mathcal{C}_{t'})$. Hence, μ' can also be considered as a mapping $\mu^* \colon \mathcal{N}(\mathcal{S}_t) \to \mathcal{N}(\mathcal{C}_t^*)$. Moreover, it is easy to verify that μ^* satisfies the properties in Definition 4. Note that with any elimination of a duplicate in step 3, we can adjust this mapping such that these properties remain intact: For the subtree that is deleted in the redundancy elimination, there is an identical subtree in the remaining s-tree. We adjust the mapping μ^* such that any node in $\mathcal{S}_t$ that has been mapped to a node in the deleted subtree is now mapped to the corresponding node in the remaining s-tree.

For the s-tree $\mathcal{C}_t$ at t that results from steps 1–3, we can therefore construct a mapping μ satisfying the properties in Definition 4, and no node in $\mathcal{C}_t$ has identical child subtrees, as redundancies have been eliminated exhaustively. Thus $\mathcal{C}_t$ is valid.

For an *introduce node* t with child t', we first give an intuition of our procedure: We obtain the valid s-tree $\mathcal{C}_t$ from $\mathcal{C}_{t'}$ by copying subtrees of the valid s-tree $\mathcal{C}_{t'}$ 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 $\ell(N) \cup \{b\}$, where b is the introduced bag element. Similarly, every node N in $\mathcal{C}_{t'}$ with $\ell(N) = undef$ 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. This is expressed in detail in Figure 6.

Lemma 3. *Consider an MSO formula ϕ, a σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$, and an introduce node $t \in \mathcal{T}$ whose child node is t'. Suppose that $\chi(t) \setminus \chi(t') = \{b\}$. Then a valid s-tree $\mathcal{C}_t$ at t can be obtained from a valid s-tree $\mathcal{C}_{t'}$ at t' as described in Figure 6. We thus construct $\mathcal{C}_t$ inductively from level 0 to level $n + 1$. Alongside every node N in $\mathcal{C}_t$, we define its "origin" $O(N)$, i.e., the node in $\mathcal{C}_{t'}$ which node N "stems from". In Figure 6 we specify how to compute the children of a node N in $\mathcal{C}_t$ from the children of $O(N)$ in $\mathcal{C}_{t'}$.*

Inductive definition of $\mathcal{C}_t$ from $\mathcal{C}_{t'}$.
Define a valid s-tree $\mathcal{C}_t$ at t from a valid s-tree $\mathcal{C}_{t'}$ at t';
every node N in $\mathcal{C}_t$ is accompanied by its "origin" $O(N)$ in $\mathcal{C}_{t'}$.

Initialization: For the root r of $\mathcal{C}_t$, define $O(r) =$ root of $\mathcal{C}_{t'}$.

Child nodes and origin $(N_i, O(N_i))$ of node N in $\mathcal{C}_t$ with origin $O(N)$:

For nodes N whose children correspond to a set variable:
$\{(N_i, N_i') \mid N_i'$ is a child of $O(N)$ with
$\big(\ell(N_i) = \ell(N_i')$ or $(\ell(N_i) = \ell(N_i') \cup \{b\})\big)\}$

For nodes N whose children correspond to an individual variable:
$\{(N_i, N_i') \mid N_i'$ is a child of $O(N)$ with
$\Big((\ell(N_i) = \ell(N_i'))$ or $\big(\ell(N_i) = b$ and $\ell(N_i') = \mathit{undef}\big)\Big)\}$

For nodes N at level n:
Let I_N denote the partial assignment at N,
let N_1' be the child of $O(N)$, and let $\ell(N_1') = (At^+, At^-)$.
Then N has a child N_1 with $O(N_1) = N_1'$ and with label $\ell(N_1) = (At_*^+, At_*^-)$ defined as follows:

$$At_*^+ = At^+ \cup \{Y_i(z_j) \mid I_N(z_j) = b \text{ and } b \in I_N(Y_i)\} \cup \{R(z_{i_1}, \ldots, z_{i_\alpha}) \mid I_N(z_{i_j}) = b \text{ for some } i_j \text{ and } I_N(z_{i_j}) \neq \mathit{undef} \text{ for all } i_j \text{ and } (I_N(z_{i_1}), \ldots, I_N(z_{i_\alpha})) \in R^{\mathcal{A}}\}$$

$$At_*^- = At^- \cup \{Y_i(z_j) \mid I_N(z_j) = b \text{ and } b \notin I_N(Y_i)\} \cup \{R(z_{i_1}, \ldots, z_{i_\alpha}) \mid I_N(z_j) = b \text{ for some } i_j \text{ and } I_N(z_j) \neq \mathit{undef} \text{ for all } i_j \text{ and } (I_N(z_{i_1}), \ldots, I_N(z_{i_\alpha})) \notin R^{\mathcal{A}}\}$$

Fig. 6. Valid s-tree at an introduce node

Proof. Let $\mathcal{S}_t$ and $\mathcal{S}_{t'}$ denote the local semantic trees at t and t', respectively. Recall that we have $\chi(\geq t) = \chi(\geq t') \cup \{b\}$. To show that $\mathcal{C}_t$ is a valid s-tree it suffices to construct a mapping $\mu\colon \mathcal{N}(\mathcal{S}_t) \to \mathcal{N}(\mathcal{C}_t)$ according to Definition 4. By assumption, the corresponding mapping $\mu'\colon \mathcal{N}(\mathcal{S}_{t'}) \to \mathcal{N}(\mathcal{C}_{t'})$ exists.

Note that the "origin" function $O(\cdot)$ associates every node in $\mathcal{C}_t$ with a node in $\mathcal{C}_{t'}$. Analogously, we now define a function $Corr(\cdot)$ which associates every node in $\mathcal{S}_t$ with a "corresponding" node in $\mathcal{S}_{t'}$. Let N be a node in $\mathcal{S}_t$ and suppose that the labels of the nodes corresponding to set variables along the path from the root to N are $(C_1, \ldots, C_p)$, and the labels of the nodes corresponding to individual variables along the path from the root to N are $(d_1, \ldots, d_q)$. Then we set $Corr(N) := N'$, where N' is the uniquely defined node in $\mathcal{S}_{t'}$ at the same level as N, s.t. the nodes corresponding to set variables in the path from the root to N' have the labels $(C_1 \setminus \{b\}, \ldots, C_p \setminus \{b\})$, and the nodes corresponding to an individual variable in the path from the root to N' have the labels $(d_1', \ldots, d_q')$ with $d_i' := \mathit{undef}$ if $d_i = b$ and $d_i' := d_i$ otherwise.

We are now ready to define a function $\mu\colon \mathcal{N}(\mathcal{S}_t) \to \mathcal{N}(\mathcal{C}_t)$ that satisfies all conditions of Definition 4. Let r_S and r_C denote the root of $\mathcal{S}_t$ and $\mathcal{C}_t$, respectively. We start

the construction of μ by setting $\mu(r_S) = r_C$ and we proceed further by induction on the level i in the trees $\mathcal{S}_t$ and $\mathcal{C}_t$.

For every $i \in \{0, \ldots, n-1\}$ we inspect every node $N_S \in \mathcal{N}(\mathcal{S}_t)$ at level i. Let N_C denote the node in $\mathcal{N}(\mathcal{C}_t)$ with $\mu(N_S) = N_C$. Now, for every child N'_S of N_S, we define $\mu(N'_S) := N'_C$, where N'_C is chosen as follows:

N'_C is a child of N_C, $O(N'_C) = \mu'(Corr(N'_S))$, and the label $\ell(N'_C)$ satisfies the following conditions:

If the $(i+1)$th variable is a set variable: $b \in \ell(N'_C)$ iff $b \in \ell(N'_S)$.

If the $(i+1)$th variable is an individual variable: $\ell(N'_C) = b$ if $\ell(N'_S) = b$; $\ell(N'_C) = \mathit{undef}$ if $\ell(N'_S) = \mathit{undef}$.

First we observe that μ is thus well-defined for every node $N_S \in \mathcal{N}(\mathcal{S}_t)$ at every level $1, \ldots, n$. Indeed, by the definition of the "origin" function $O(\cdot)$ in Figure 6, we have that, for a given node $N_C \in \mathcal{N}(\mathcal{C}_t)$, there can be at most 2 child nodes N_1, N_2 of N_C with identical function values under $O(\cdot)$:

- For N_C whose children correspond to a set variable: A pair (N_1, N_2) of child nodes of N_C satisfies $O(N_1) = O(N_2)$ if and only if $\ell(N_1) = \ell(N_2) \setminus \{b\}$ or vice versa holds.
- For N_C whose children correspond to an individual variable: Exactly one pair (N_1, N_2) of child nodes of N_C satisfies $O(N_1) = O(N_2)$, namely the pair of nodes with $\ell(N_1) = \mathit{undef}$ and $\ell(N_2) = b$ or vice versa.

In other words, the identification of the parent N_C of N'_C together with the above case distinction over the label of $\ell(N'_S)$ uniquely defines the node N'_C to which N'_S is mapped by μ. Moreover, it is easy to check that μ fulfills all conditions from Definition 4 for the nodes at levels $1, \ldots n$.

We have defined μ for all nodes $N_S \in \mathcal{S}_t$ at level n and, therefore implicitly also for the nodes on level $n+1$, namely, $\mu(N'_S) = N'_C$, where N_S is the parent of N'_S, N_C is the parent of N'_C, and $\mu(N_S) = N_C$. It remains to verify that the labels of N'_S and N'_C indeed coincide.

We have to show that the computation of $\ell(N_C)$ from $\ell(O(N_C))$ for nodes N_C at level $n+1$ according to Figure 6 yields the correct result. To this end, we inspect how $\ell(N_S)$ can be computed from $\ell(Corr(N_S))$, where N_S is at level $n+1$: Let I denote the truth assignment that is given by the labels $(C_1, \ldots, C_m)$ of the nodes corresponding to set variables along the path from the root of $\mathcal{S}_t$ to N_S, as well as by the labels $(d_1, \ldots, d_k)$ of the nodes corresponding to individual variables along that path. Furthermore, let I' denote the truth assignment given by the labels $(C_1 \setminus \{b\}, \ldots, C_m \setminus \{b\})$ of the nodes corresponding to set variables along the path from the root of $\mathcal{S}_{t'}$ to $Corr(N_S)$, and by the labels $(d'_1, \ldots, d'_k)$ of the nodes corresponding to individual variables, where $d'_i := \mathit{undef}$ if $d_i = b$ and $d'_i := d_i$ otherwise.

Suppose an atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is true in I. Then $d_{i_j} \in \chi(\geq t)$ holds for all j and $(d_{i_1}, \ldots, d_{i_\alpha}) \in R^{\mathcal{A}}$. We distinguish the following cases:

1. If $d_{i_j} \in \chi(\geq t) \setminus \{b\}$ for all j then $R(z_{i_1}, \ldots, z_{i_\alpha})$ is already true in I'.
2. If $d_{i_j} = b$ for at least one j then, by the definition of tree decompositions, we actually have $d_{i_j} \in \chi(t)$ for all j. Moreover, by the definition of $Corr(\cdot)$, the truth value of $R(z_{i_1}, \ldots, z_{i_\alpha})$ in I' is undefined.

Suppose an atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I. Then $d_{i_j} \in \chi(\geq t)$ for all j and $(d_{i_1}, \ldots, d_{i_\alpha}) \notin R^{\mathcal{A}}$. We distinguish the following cases:

1. If $d_{i_j} \in \chi(\geq t) \setminus \{b\}$ for all j then $R(z_{i_1}, \ldots, z_{i_\alpha})$ is already false in I'.
2. If $d_{i_j} = b$ for at least one j then the truth value of this atom is undefined in I'.

Suppose an atom $Y_i(z_j)$ is true in I. Then $d_j \in \chi(\geq t)$ and $d_j \in C_i$. We distinguish the following cases:

1. If $d_j \in \chi(\geq t) \setminus \{b\}$ then $Y_i(z_j)$ is already true in I'.
2. If $d_j = b$ then the truth value of $Y_i(z_j)$ is undefined in I'.

Suppose an atom $Y_i(z_j)$ is false in I. Then $d_j \in \chi(\geq t)$ and $d_j \notin C_i$. We distinguish the following cases:

1. If $d_j \in \chi(\geq t) \setminus \{b\}$ then $Y_i(z_j)$ is already false in I'.
2. If $d_j = b$ then the truth value of this atom is undefined in I'.

To check that the sets At^+_* and At^-_* in Figure 6 have the correct values, we have to inspect the various cases above:

- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is true in I if either (case 1 above) it is already true in I' or (case 2 above) $d_j = b$ for some j, $d_{i_j} \in \chi(t)$ for all j, and $(d_{i_1}, \ldots, d_{i_\alpha}) \in R^{\mathcal{A}}$. Since $d_{i_j} \in \chi(t)$ for all j, this is correctly checked in the computation of At^+_* in Figure 6.
- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I if either (case 1 above) it is already false in I' or (case 2 above) $d_{i_j} = b$ for some j and $(d_{i_1}, \ldots, d_{i_\alpha}) \notin R^{\mathcal{A}}$. For the purpose of checking $(d_{i_1}, \ldots, d_{i_\alpha}) \notin R^{\mathcal{A}}$ in Figure 6, we may treat $\star$ in any position j' like an ordinary domain element. Of course, the test $(I_N(z_{i_1}), \ldots, I_N(z_{i_\alpha})) \notin R^{\mathcal{A}}$ in Figure 6 is guaranteed to succeed whenever $I_N(z_{i_{j'}}) = \star$ for some j'. This is the correct behavior since $\star$ stands for some value $d_{i_{j'}} \in \chi(> t)$. Thus, in the computation of At^-_* in Figure 6, we check correctly if $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I. Note that, by the definition of valid s-trees, At^- may contain atoms $R(z_{i_1}, \ldots, z_{i_\alpha})$ which were undefined in I'. Of course, this can only be the case if all previously undefined variables of such an atom are now interpreted as b in I. Recall that $\star$ stands for a domain value in $\chi(> t)$. But then, by the connectedness condition, the tree decomposition $\mathcal{T}$ cannot have a node whose bag jointly contains all the elements $I(z_{i_1}), \ldots, I(z_{i_\alpha})$. In other words, such an atom must have the truth value false in I. Hence, again it is correct to have all these atoms in At^-_*.
- An atom $Y_i(z_j)$ is true in I if either (case 1 above) it is already true in I' or we have $d_j = b$ and $b \in C_i$. These conditions are correctly checked in the computation of At^+_* in Figure 6.
- An atom $Y_i(z_j)$ is false in I if either (case 1 above) it is already false in I' or we have $d_j = b$ and $b \notin C_i$. These conditions are correctly checked in the computation of At^-_* in Figure 6.
- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ or $Y_i(z_j)$ is undefined in I if it was undefined in I' and if its truth value is not set to either true or false in one of the cases analyzed above. Hence, in Figure 6, we correctly let all atoms undefined that are neither included in At^+_* nor in At^-_*.

To conclude, At^+_* and At^-_* in Figure 6 indeed correctly identify the sets of atoms from ϕ that are either true or false in I. We have thus established the existence of a mapping μ as required by Definition 4. Furthermore, at introduce nodes no redundancies in the form of identical sibling subtrees can arise because $\mathcal{C}_{t'}$ does not contain any and it can easily be seen that none of our modifications to construct $\mathcal{C}_t$ introduce redundancies. This proves that $\mathcal{C}_t$ is valid.

Now we give the intuition of our procedure for a *join node* t 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 $\mathcal{C}_{t_2}$. A node N_1 in $\mathcal{C}_{t_1}$ and a node N_2 in $\mathcal{C}_{t_2}$ are compatible if either they correspond to the same set variable and $\ell(N_1) = \ell(N_2)$, or they correspond to the same individual variable and the following holds: 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 $\ell(N_i) \neq undef$. 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 as follows. We set At^+ to the union of the true atoms in the corresponding nodes of $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$. The set of false atoms At^- consists of the union of the false atoms in the corresponding nodes of $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$, but it may also contain atoms $R(z_{i_1}, \ldots, z_{i_\alpha})$ that were neither true nor false in either of these nodes. This is the case if our combination of compatible nodes in $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$ leads to no z_{i_j} having the value $undef$ anymore. This is formalized in Figure 7.

Lemma 4. *Consider an MSO formula ϕ, a σ-structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$, and a join node $t \in \mathcal{T}$ whose child nodes are t_1 and t_2. Then a valid s-tree $\mathcal{C}_t$ at t can be obtained from valid s-trees $\mathcal{C}_{t_1}$ at t_1 and $\mathcal{C}_{t_2}$ at t_2 as described in Figure 7: We construct $\mathcal{C}_t$ inductively from level 0 to level $n+1$. Alongside every node N in $\mathcal{C}_t$, we define its "origins" $(O_1(N), O_2(N))$, i.e., a node $O_1(N)$ in $\mathcal{C}_{t_1}$ and a node $O_2(N)$ in $\mathcal{C}_{t_2}$, s.t. node N "stems from" $O_1(N)$ and $O_2(N)$. In Figure 7 we specify how to compute the children of a node N in $\mathcal{C}_t$ from the children of $O_1(N)$ in $\mathcal{C}_{t_1}$ and $O_2(N)$ in $\mathcal{C}_{t_2}$.*

Proof. The proof proceeds by the same steps as the proof of Lemma 3. Let $\mathcal{S}_t$, $\mathcal{S}_{t_1}$ and $\mathcal{S}_{t_2}$ denote the local semantic tree at t, t_1 and t_2, respectively. Recall that we have $\chi(t) = \chi(t_1) = \chi(t_2)$. To show that $\mathcal{C}_t$ is a valid s-tree it suffices to construct a mapping $\mu\colon \mathcal{N}(\mathcal{S}_t) \to \mathcal{N}(\mathcal{C}_t)$ according to Definition 4. By Definition 4, we know that the corresponding mappings $\mu_1\colon \mathcal{N}(\mathcal{S}_{t_1}) \to \mathcal{N}(\mathcal{C}_{t_1})$ and $\mu_2\colon \mathcal{N}(\mathcal{S}_{t_2}) \to \mathcal{N}(\mathcal{C}_{t_2})$ exist.

Note that the "origin" functions O_1 and O_2 associate every node in $\mathcal{C}_t$ with a node in $\mathcal{C}_{t_1}$ and a node in $\mathcal{C}_{t_2}$, respectively. Analogously, we now define functions $Corr_1(\cdot)$ and $Corr_2(\cdot)$ which associate every node in $\mathcal{S}_t$ with "corresponding" nodes in $\mathcal{S}_{t_1}$ and $\mathcal{S}_{t_2}$, respectively. Let N be a node in $\mathcal{S}_t$ and suppose that the labels of the nodes corresponding to set variables along the path from the root to N are $(C_1, \ldots, C_p)$, and the labels of the nodes corresponding to individual variables along that path are $(d_1, \ldots, d_q)$. Then we set $Corr_1(N) := N_1$ and $Corr_2(N) := N_2$, where N_1 and N_2 are the uniquely

Inductive definition of $\mathcal{C}_t$ from $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$.
Define a valid s-tree $\mathcal{C}_t$ at t from valid s-trees $\mathcal{C}_{t_1}$ at t_1 and $\mathcal{C}_{t_2}$ at t_2; every node N in $\mathcal{C}_t$ is accompanied by its "origins" $O_1(N)$ in $\mathcal{C}_{t_1}$ and $O_2(N)$ in $\mathcal{C}_{t_2}$.

Initialization: For the root r of $\mathcal{C}_t$, define $O_1(r) =$ root of $\mathcal{C}_{t_1}$ and $O_2(r) =$ root of $\mathcal{C}_{t_2}$.

Child nodes and origins $(N', O_1(N'), O_2(N'))$ of node N in $\mathcal{C}_t$ with origin $O_1(N)$ in $\mathcal{C}_{t_1}$ and $O_2(N))$ in $\mathcal{C}_{t_2}$:

For nodes N whose children correspond to a set variable:
$\{(N', O_1(N'), O_2(N')) \mid$ there exist N_1 child of $O_1(N)$ and N_2 child of $O_2(N)$ with $\ell(N_1) = \ell(N_2)$, s.t. $O_1(N') = N_1$ and $O_2(N') = N_2$ and $\ell(N') = \ell(N_1)\}$

For nodes N whose children correspond to an individual variable:
$\{(N', O_1(N'), O_2(N')) \mid$ there exist N_1 child of $O_1(N)$ and N_2 child of $O_2(N)$ with $\ell(N_1) = \ell(N_2)$ but $\ell(N_1) \neq \star$, or
$\ell(N_1) = \mathit{undef}$ or $\ell(N_2) = \mathit{undef}$, s.t.
$O_1(N') = N_1$ and $O_2(N') = N_2$ with
$\ell(N') = \ell(N_1)$ if $\ell(N_1) \neq \mathit{undef}$ and $\ell(N') = \ell(N_2)$ otherwise$\}$.

For nodes N at level n:
Let I_N denote the partial assignment at N and let $N_1 = O_1(N)$ with child N_1' and $N_2 = O_2(N)$ with child N_2';
moreover, let $\ell(N_1') = (At_1^+, At_1^-)$ and $\ell(N_2') = (At_2^+, At_2^-)$;
then N has a child N' with label $\ell(N') = (At_*^+, At_*^-)$, s.t.
$At_*^+ = At_1^+ \cup At_2^+$ and
$At_*^- = At_1^- \cup At_2^- \cup$
$\{R(z_{i_1}, \ldots, z_{i_\alpha}) \mid \ell(z_{i_j}) \neq \mathit{undef}$ for all i_j
and $R(z_{i_1}, \ldots, z_{i_\alpha}) \notin At_*^+\}$

Fig. 7. Valid s-tree at a join node

defined nodes in $\mathcal{S}_{t_1}$ and $\mathcal{S}_{t_2}$, respectively, at the same level as N, s.t. the nodes corresponding to set variables in the paths from the respective root to N_1 as well as N_2 have the labels $(C_1, \ldots, C_p)$; furthermore, the nodes corresponding to individual variables in the path from the root to N_1 have the labels $(d_1', \ldots, d_q')$, and the nodes corresponding to individual variables in the path from the root to N_2 have the labels $(d_1'', \ldots, d_q'')$ s.t.

if $d_i \in \chi(t) \cup \{\mathit{undef}\}$ then $d_i' := d_i$ and $d_i'' := d_i$,
if $d_i \in \chi(> t_1)$ then $d_i' := d_i$ and $d_i'' := \mathit{undef}$, and
if $d_i \in \chi(> t_2)$ then $d_i'' := d_i$ and $d_i' := \mathit{undef}$.

We are now ready to define a function $\mu\colon \mathcal{N}(\mathcal{S}_t) \to \mathcal{N}(\mathcal{C}_t)$ and we show that it satisfies all conditions of Definition 4. Let r_S and r_C denote the roots of $\mathcal{S}_t$ and $\mathcal{C}_t$, respectively. We start the construction of μ by setting $\mu(r_S) = r_C$ and we proceed further by induction on the level i in the trees $\mathcal{S}_t$ and $\mathcal{C}_t$.

For every $i \in \{0, \ldots, n-1\}$ we inspect every node $N_S \in \mathcal{N}(\mathcal{S}_t)$ at level i. Let N_C denote the node in $\mathcal{N}(\mathcal{C}_t)$ with $\mu(N_S) = N_C$. Now, for every child N_S' of N_S, we define $\mu(N') := N_C'$, where N_C' is chosen as follows: N_C' is a child of N_C, $O_1(N_C') = \mu_1(\mathit{Corr}_1(N_S'))$, and $O_2(N_C') = \mu_2(\mathit{Corr}_2(N_S'))$. Note that μ is thus well-defined for every node $N_S \in \mathcal{N}(\mathcal{S}_t)$ at every level $1, \ldots, n$ since, by the definition of the "origin"

functions $O_1(\cdot)$ and $O_2(\cdot)$ in Figure 7, the node N'_C is uniquely defined by the parent N_C of N'_C and the "origin" nodes $O_1(N'_C)$ and $O_2(N'_C)$.

We have defined μ for all nodes $N_S \in \mathcal{S}_t$ at level n and, therefore implicitly also for the nodes on level $n+1$, namely, $\mu(N'_S) = N'_C$, where N_S is the parent of N'_S, N_C is the parent of N'_C, and $\mu(N_S) = N_C$. It remains to verify that the labels of N'_S and N'_C indeed coincide.

We have to show that the computation of $\ell(N_C)$ from $\ell(O_1(N_C))$ and $\ell(O_2(N_C))$ for nodes N_C at level $n+1$ according to Figure 7 yields the correct result. To this end, we inspect how $\ell(N_S)$ can be computed from $\ell(Corr_1(N_S))$ and $\ell(Corr_2(N_S))$, where N_S is at level $n+1$: Let I denote the truth assignment that is given by the labels $(C_1, \ldots, C_m)$ of the nodes corresponding to set variables along the path from the root of $\mathcal{S}_t$ to N_S, as well as by the labels $(d_1, \ldots, d_k)$ of the nodes corresponding to individual variables along that path. Furthermore, let I_1 and I_2 denote the truth assignments given by the labels $(C'_1, \ldots, C'_m)$ and $(C''_1, \ldots, C''_m)$ of the nodes corresponding to set variables along the path from the roots of $\mathcal{S}_{t_1}$ and $\mathcal{S}_{t_2}$ to $Corr_1(N_S)$ and $Corr_2(N_S)$, respectively, and the labels $(d'_1, \ldots, d'_k)$ and $(d''_1, \ldots, d''_k)$ of the nodes corresponding to individual variables along the respective path. Recall that by the above definition of $Corr_1(\cdot)$ and $Corr_2(\cdot)$, the sets C'_i, C''_i satisfy the conditions $C'_i := C_i \cap \chi(\geq t_1)$ and $C''_i := C_i \cap \chi(\geq t_2)$. Moreover, the domain elements d'_i, d''_i satisfy the following conditions:

If $d_i \in \chi(t) \cup \{undef\}$, then $d'_i := d_i$ and $d''_i := d_i$
If $d_i \in \chi(> t_1)$, then $d'_i := d_i$ and $d''_i := undef$, and
If $d_i \in \chi(> t_2)$, then $d''_i := d_i$ and $d'_i := undef$.

Suppose an atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is true in I. Then $d_{i_j} \in \chi(\geq t)$ holds for all j and $(d_{i_1}, \ldots, d_{i_\alpha}) \in R^{\mathcal{A}}$. We distinguish the following cases:

1. If $d_{i_j} \in \chi(t)$ for all j, then also $d_{i_j} \in \chi(\geq t_1)$ and $d_{i_j} \in \chi(\geq t_2)$ for all j. Hence, $R(z_{i_1}, \ldots, z_{i_\alpha})$ is already true in both I_1 and I_2.
2. If $d_{i_j} \in \chi(> t_1)$ for at least one j then, by the definition of tree decompositions, we actually have $d_{i_j} \in \chi(\geq t_1)$ for all j. Hence, $R(z_{i_1}, \ldots, z_{i_\alpha})$ is already true in I_1. On the other hand, by the definition of $Corr_1(\cdot)$ and $Corr_2(\cdot)$, $R(z_{i_1}, \ldots, z_{i_\alpha})$ is undefined in I_2.
3. The case $d_{i_j} \in \chi(> t_2)$ for at least one j is symmetric to case 2 above.

Suppose an atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I. Then $d_{i_j} \in \chi(\geq t)$ for all j and $(d_{i_1}, \ldots, d_{i_\alpha}) \notin R^{\mathcal{A}}$. We distinguish the following cases:

1. If $d_{i_j} \in \chi(t)$ for all j, then also $d_{i_j} \in \chi(\geq t_1)$ and $d_{i_j} \in \chi(\geq t_2)$ for all j. Hence, $R(z_{i_1}, \ldots, z_{i_\alpha})$ is already false in both I_1 and I_2.
2. If $d_{i_j} \in \chi(\geq t_1)$ for all j and $d_{i_{j'}} \in \chi(> t_1)$ for some j', then $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I_1 and undefined in I_2.
3. The case $d_{i_j} \in \chi(\geq t_2)$ for all j and $d_{i_{j'}} \in \chi(> t_2)$ for some j' is symmetric to case 2 above.
4. If $d_{i_j} \in \chi(> t_1)$ for some j and $d_{i_{j'}} \in \chi(> t_2)$ for some j', then $R(z_{i_1}, \ldots, z_{i_\alpha})$ is undefined in both I_1 and I_2.

Suppose an atom $Y_i(z_j)$ is true in I. Then $d_j \in \chi(\geq t)$ and $d_j \in C_i$. We distinguish the following cases:

1. If $d_j \in \chi(t)$, then we also have $d_j \in \chi(\geq t_1)$ and $d_j \in \chi(\geq t_2)$. Hence, $Y_i(z_j)$ is already true in both I_1 and I_2.
2. If $d_j \in \chi(> t_1)$, then we claim that $d_j \in C'_i$. Indeed, by the definition of tree decompositions, from $d_j \in \chi(> t_1)$ it follows that $d_j \notin \chi(\geq t_2)$. Hence, from $d_j \in C_i$ and $C_i = C'_i \cup C''_i$ we conclude that $d_j \in C'_i$. But then $Y_i(z_j)$ is already true in I_1. On the other hand, $Y_i(z_j)$ is undefined in I_2.
3. The case $d_j \in \chi(> t_2)$ is symmetric to case 2 above.

Suppose an atom $Y_i(z_j)$ is false in I. Then $d_j \in \chi(\geq t)$ and $d_j \notin C_i$. We distinguish the following cases

1. If $d_j \in \chi(t)$, then we also have $d_j \in \chi(\geq t_1)$ and $d_j \in \chi(\geq t_2)$. Hence, $Y_i(z_j)$ is already false in both I_1 and I_2.
2. If $d_j \in \chi(> t_1)$, then $Y_i(z_j)$ is false in I_1 and undefined in I_2.
3. The case $d_j \in \chi(> t_2)$ is symmetric to case 2 above.

To check that the sets At^+_* and At^-_* in Figure 7 have the correct values, we have to inspect the various cases above:

- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is true in I if either it is already true in I_1 (cases 1 and 2 above) or in I_2 (cases 1 and 3 above). In Figure 7 we thus correctly put $R(z_{i_1}, \ldots, z_{i_\alpha})$ into At^+_*.
- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ is false in I if it is false in both I_1 and I_2 (case 1 above), or it is false in one of $\{I_1, I_2\}$ and undefined in the other (cases 2 and 3 above). These cases are correctly checked in the computation of At^-_* in Figure 7.
  Now consider those atoms that are false in I and whose truth value was undefined in I_1 and I_2 (case 4 above). Those atoms are correctly added to the set At^-_* in Figure 7 because the set

  $$\{R(z_{i_1}, \ldots, z_{i_\alpha}) \mid \ell(z_{i_j}) \neq \mathit{undef} \text{ for all } i_j \text{ and } R(z_{i_1}, \ldots, z_{i_\alpha}) \notin At^+_*\}$$

  consists only of those atoms as well as atoms that are false in I_1 or I_2.
- An atom $Y_i(z_j)$ is true in I if either (cases 1 and 2 above) it is true in I_1 or (cases 1 and 3 above) it is true in I_2. In Figure 7 we thus correctly put $Y_i(z_j)$ into At^+_*.
- An atom $Y_i(z_j)$ is false in I if either (cases 1 and 2 above) it is false in I_1 or (cases 1 and 3 above) it is false in I_2 In Figure 7 we thus correctly put $Y_i(z_j)$ into At^-_*.
- An atom $R(z_{i_1}, \ldots, z_{i_\alpha})$ or $Y_i(z_j)$ is undefined in I if it was undefined in both I_1 and I_2 and if its truth value is not set to either true or false in one of the cases analyzed above. Hence, in Figure 7, we correctly let all atoms undefined that are neither included in At^+_* nor in At^-_*.

To conclude, At^+_* and At^-_* in Figure 7 indeed correctly identify the sets of atoms from ϕ that are true or, respectively, false in I

2.4 MSO Model Checking via s-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, in a bottom-up traversal of $\mathcal{T}$, we compute an s-tree at every node in $\mathcal{T}$. Then we evaluate ϕ over $\mathcal{A}$ by reducing the s-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 a valid s-tree $\mathcal{C}_t$ at every node t in $\mathcal{T}$ in overall time $O(f(\tau(\mathcal{T}), \phi) \cdot ||\mathcal{T}||)$. Here, $\tau(\mathcal{T})$ denotes the width of $\mathcal{T}$ and f is a function not depending on $\mathcal{A}$.*

Proof. In a bottom-up traversal of $\mathcal{T}$ we proceed as described in Lemmas 1 – 4. To prove an upper bound on the complexity of the computation at each node, we prove an upper bound on the size of valid s-trees. It is then easy to see that the computation of a valid s-tree $\mathcal{C}_t$ at every node $t \in \mathcal{T}$ can be done within the time bound stated in the theorem.

We can easily prove by an induction argument that the number of nodes in an s-tree is bounded by a tower of exponentials whose height corresponds to the quantifier rank n of ϕ. For a node corresponding to an individual variable, the number of possible labels is bounded by $w + 3$ ($w + 1$ bag elements plus $undef$ and $\star$). For a node corresponding to a set variable, this number is 2^{w+1}. Supposing w.l.o.g. that $w > 0$, the number of possible labels of an inner node is then bounded by 2^{w+1}. A node at level $n - 1$ can therefore have up to $2^{w+1} \cdot 3^{||\phi||}$ children, as each atom can either be in At^+, At^- or neither. At level $n - 2$, a node can have at most $2^{w+1} \cdot 2^{2^{w+1} \cdot 3^{||\phi||}}$ children.

By iterating these ideas, we can compute an upper bound on the number of child nodes of nodes at levels $n - 3, \ldots, 1, 0$. We thus get an expression which is a tower of exponentials whose height corresponds to the quantifier rank of ϕ. But of course, the expression does not depend on the size of structure $\mathcal{A}$. The size of the valid s-tree is thus also $O(g(\tau(\mathcal{T}), \phi))$ for some function g that depends on the width of $\mathcal{T}$ and ϕ but not on $\mathcal{A}$.

Now suppose a naive implementation of the computations described in Lemmas 1 – 4: We iterate in (nested) loops over the nodes of the s-tree at the child node(s) of t. Also for the reduction of the resulting s-tree at t, we simply proceed in (nested) loops. Clearly, the complexity of all these operations depends solely on $\tau(\mathcal{T})$ and ϕ but not on $\mathcal{A}$.

3 ASP and D-FLAT

In this section, we give brief introductions to Answer Set Programming (ASP) [8] and the D-FLAT system [2, 1, 4]. 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*

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

The constituents of a rule $r \in \Pi$ are $h(r) = \{a_1, \ldots, a_k\}$, $b^+(r) = \{b_1, \ldots, b_m\}$ and $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. 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 if $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a *model* of a set of rules if it satisfies each rule. I is an *answer set* of a program Π if it is a subset-minimal model of the program $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$ [20].

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* [17, 18] (version 4) 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 4. The following ASP 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.

Informally, the first rule 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 adjacent to some vertex in this set.

In order to take advantage of this Guess & Check approach in a decomposed setting, we make use of the D-FLAT system [2, 1, 4]. 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 8. 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

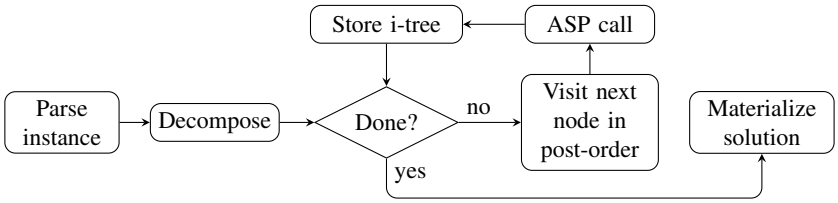

Fig. 8. Control flow in D-FLAT

us to obtain complete solutions by combining the item sets reachable by recursively following extension pointers. (This is very similar to the "origin" function that we used in Lemmas 3 and 4 to associate an s-tree node with the nodes it stems from.) These notions can be formalized as follows.

Definition 5. *Let $\mathcal{T}$ be a tree decomposition and let $t \in \mathcal{T}$. The* i-tree *of t is a triple (S_t, X_t, Y_t) where S_t is a rooted tree. When it is clear from the context, we sometimes use the term "i-tree" to denote only S_t instead of (S_t, X_t, Y_t). The function X_t assigns to each* i-tree node $N \in S_t$ *an* item set*, where each item is some arbitrary string. Let $t_1, \ldots, t_k \in \mathcal{T}$ be the child nodes of t and let N be an i-tree node in S_t. Then Y_t is a function that assigns to N a non-empty set of tuples called* extension pointers *such that the following properties hold. If $N, N_1, \ldots, N_k$ are the root nodes of $S_t, S_{t_1}, \ldots, S_{t_k}$, respectively, then $Y_t(N) = \{(N_1, \ldots, N_k)\}$. If N has a parent N' then for each $(N_1, \ldots, N_k) \in Y_t(N)$ there is some $(N'_1, \ldots, N'_k) \in Y_t(N')$ such that each N_i is a child of N'_i in S_{t_i}, and for each $(N'_1, \ldots, N'_k) \in Y_t(N')$ there is some $(N_1, \ldots, N_k) \in Y_t(N)$ such that each N_i is a child of N'_i in S_{t_i}.*

Note that all nodes in an i-tree at a leaf of a tree decomposition have exactly one extension pointer, namely the empty tuple. Furthermore, for any tree decomposition node t with children $t_1, \ldots, t_k$, the root node N of S_t also has exactly one extension pointer, namely $(N_1, \ldots, N_k)$, where any N_i is the root of S_{t_i}. The construction of complete solutions by following extension pointers can be formalized as follows.

Definition 6. *Let $\mathcal{T}$ be a tree decomposition and (S_t, X_t, Y_t) be the i-tree at a node $t \in \mathcal{T}$. Moreover, let $t_1, \ldots, t_k \in \mathcal{T}$ be the child nodes of t. We inductively define the set of* extensions *of an i-tree node $N \in S_t$ as $Z(N) = \{\{N\} \cup A \mid A \in \bigcup_{(N_1,\ldots,N_k)\in Y_t(N)} \{A_1 \cup \cdots \cup A_k \mid A_i \in Z(N_i)$ for all $1 \leq i \leq k\}\}$.*

Example 5. For the INDEPENDENT DOMINATING SET problem, we can use i-trees to represent partial solutions in the following way: At each node $t \in \mathcal{T}$, where $\mathcal{T}$ is a tree decomposition of the input, we store an i-tree of height 1 such that the root item set remains empty and each leaf item set stores the status of elements of the current bag $\chi(t)$. We can store the status of a vertex $v \in \chi(t)$ by means of the following items: If the vertex v has been guessed to be contained in the partial solution, we store an item $\texttt{in}(v)$. If there is a vertex $w \in \chi(t)$ such that w dominates v (i.e., w is "in" and there is an edge from w to v), then we store an item $\texttt{dominated}(v)$. Otherwise, we do not store an item containing v.

Formally, let $t_1, \ldots, t_k$ be the children of t in $\mathcal{T}$. First we define the root of S_t to be a node N such that $X_t(N) = \emptyset$ and $Y_t(N) = \{(N_1, \ldots, N_k)\}$, where each N_i is the

root of S_{t_i}. Now for any node $N_i \in t_i$, let $I_i(N_i) = \{v \in \chi(t_i) \mid \texttt{in}(v) \in X_{t_i}(N_i)\}$ and $D_i(N_i) = \{v \in \chi(t_i) \mid \texttt{dominated}(v) \in X_{t_i}(N_i)\}$. For any $I \subseteq \chi(t)$ and $D \subseteq \chi(t) \setminus I$, we say that a tuple of leaves $(N_1, \ldots, N_k)$ from $S_{t_1}, \ldots, S_{t_k}$, respectively, is *compatible* with I and D if $I_i(N_i) \cap \chi(t) = I \cap \chi(t_i)$ and $D = D_1(N_1) \cup \cdots \cup D_k(N_k) \cup \{v \in \chi(t) \mid w \in I \text{ and there is an edge } (w, v)\}$. Then we add to the root of S_t a child N such that $X_t(N) = \{\texttt{in}(v) \mid v \in I\} \cup \{\texttt{dominated}(v) \mid v \in D\}$ and $Y_t(N)$ is the set of tuples $(N_1, \ldots, N_k)$ such that each N_i is a leaf in S_{t_i} and $(N_1, \ldots, N_k)$ is compatible with I and D.

By using extension pointers that associate each i-tree node with predecessor nodes from the i-trees at the children of t in this way, we can combine the information in the item sets that we can reach recursively via extension pointers. This way, we obtain a partial solution where each vertex occurring in a bag of the subtree rooted at t is assigned a status as described above. When t is the root, the extensions of the leaves of S_t therefore encode exactly the complete solutions because the bags in the subtree rooted at t comprise all vertices of the input graph.

As input to the user's problem-specific ASP program, D-FLAT declares the following facts.

- `final` if the current node $t \in \mathcal{T}$ is the root of $\mathcal{T}$.
- `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.
- `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 of the user's ASP program together with this input 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`(d, j) causes that j is added to the extension pointers of the node at depth d of the branch.
- `item`(d, i) states that the node at depth d of the branch contains i in its item set.

These are the main predicates that allow D-FLAT to construct an i-tree at each tree decomposition node. Once it has computed such a tree at each node, it remains to decide the problem. D-FLAT does this by inspecting the i-tree at the root of the tree decomposition: In analogy to Alternating Turing Machines [10], D-FLAT maps each inner node of this i-tree to either "or" or "and", and each leaf to either "accept" or "reject". The mapping in D-FLAT, however, is partial: Sometimes it only becomes clear at some later point in the tree decomposition which status an i-tree node should be mapped to. In order to specify this mapping, atoms of the following form are recognized by D-FLAT in the answer sets:

- If an atom $\texttt{or}(i)$ or $\texttt{and}(i)$ is present in an answer set, then we map node at depth i of the branch specified by that answer set to "or" or "and", respectively.
- If an atom `accept` or `reject` is present in an answer set, then we map the leaf of the branch specified by that answer set to "accept" or "reject", respectively.

With this mapping, D-FLAT can propagate acceptance statuses from the leaves to the root of the i-tree. Before that, however, in case the current decomposition node is the root, D-FLAT first prunes all subtrees of this i-tree that are rooted at a node not mapped to any of those statuses.

After mapping i-tree nodes to statuses "or", "and", "accept" or "reject" in this manner (and deleting nodes with undefined status if the current decomposition node is the root), D-FLAT propagates acceptance statuses from the leaves to the root of the i-tree in the following way. We call an i-tree node *accepting* if it has either been mapped to "accept", or it has been mapped to "or" and some child is accepting, or it has been mapped to "and" and all children are accepting. On the other hand, we call an i-tree node *rejecting* if it has either been mapped to "reject", or it has been mapped to "or" and all children are rejecting, or it has been mapped to "and" and some child is rejecting. Nodes that are neither accepting nor rejecting are said to have an *undefined* acceptance status. After an i-tree has been computed and acceptance statuses have been propagated, D-FLAT prunes subtrees from the i-tree that have been found to be rejecting in order to avoid unnecessary computations. For a given D-FLAT encoding, we now say that D-FLAT *accepts* an input structure $\mathcal{A}$ with tree decomposition $\mathcal{T}$ if the i-tree at the root of $\mathcal{T}$ has an accepting root node.

Example 6. For INDEPENDENT DOMINATING SET, the relevant part of an answer set could look like this:

$$\{\texttt{length}(1), \texttt{or}(0), \texttt{accept},\\ \texttt{extend}(0, n_1), \texttt{extend}(1, n_2), \texttt{item}(1, \texttt{in}(v_1)), \texttt{item}(1, \texttt{dominated}(v_2))\}$$

This answer set would lead to a branch of length 1 in the i-tree such that the leaf contains the two specified items. The root and leaf would have as predecessors n_1 and n_2 (nodes from the child i-tree), respectively. The `accept` atom indicates that this i-tree node, together with its predecessors reachable via extension pointers, witnesses a solution. It will cause the i-tree root, which is mapped to "or", to be accepting.

All atoms using `extend`, `item`, `or` and `and` with the same depth argument, as well as `accept` and `reject`, constitute what we call a *node specification*. To determine where branches diverge, D-FLAT uses the following recursive condition: Two node specifications (potentially from different answer sets) coincide (i.e., describe the same i-tree node) iff

(1) they are at the same depth in the i-tree,
(2) their item sets, extension pointers and statuses ("and", "or", "accept" or "reject") are equal, and
(3) 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 and statuses. 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 7. Listing 1.1 shows a D-FLAT encoding for INDEPENDENT DOMINATING SET. All i-tree branches have height 1 and the roots are "or" nodes (due to line 1); the item sets of the roots are always empty and item sets at leaves contain items involving the function symbols `in` and `dominated`. Note that lines 7–10 resemble the program from Example 4, 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.[4] 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 a vertex called 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". Note that we require the root of the tree decomposition to have an empty bag, so by the time the root is reached all vertices have been considered by this constraint. 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".

Finally, line 13 ensures that, if the currently processed node is the tree decomposition's root, the leaf is mapped to "accept", leading to D-FLAT accepting the input if there are any answer sets at the tree decomposition's root. Note that any of these answer sets contains the atom `accept`. This is correct because invalid solution candidates are eliminated in lines 8 and 10 and thus do not produce answer sets.

[4] It should be noted that D-FLAT works on all tree decompositions, not only the so-called normalized ones, as used in this paper, that require each node to be a leaf, introduce, forget or join node.

```
length(1). or(0).                                                    1
extend(0,R) ← root(R).                                               2
1 { extend(1,S) : sub(R,S) } 1 ← extend(0,R).                        3
← extend(1,S;1,T), childItem(S,in(X)),                               4
  not childItem(T,in(X)).
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))        ← current(X), in(X).                            11
item(1,dominated(X)) ← current(X), dominated(X).                     12
accept ← final.                                                      13
```

Listing 1.1. Computing independent dominating sets with D-FLAT

4 MSO Model Checking on Tree Decompositions with ASP

We now present a D-FLAT encoding solving MSO MC in the style of the approach from Section 2 in order to show that D-FLAT allows us to solve any MSO-definable problem in linear time for bounded treewidth. The basic idea is to use i-trees for representing s-trees and proceed like in Lemmas 1 – 4 for the bottom-up computation.

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 we would like to know whether $\mathcal{A} \models \phi$ holds, to be in prenex normal form. Here we additionally assume that its matrix is in CNF. Our encoding can, however, be easily generalized. Much could be done to improve the MSO model checker that emerges from this work; but this is outside the scope of this paper, as we focus on the general applicability of D-FLAT.

The formula ϕ is specified in ASP as follows. If the quantifier rank is r, then the fact $\texttt{length}(r{+}1)$ is declared. (This will cause each i-tree branch to consist of $r{+}2$ nodes.) 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.[5] 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. The atoms appearing as the second argument to `pos` and `neg` are represented as terms of the form $\texttt{in}(x, X)$ (denoting the atom $x \in X$) or $\texttt{edge}(x, y)$ (denoting the atom that expresses membership in the edge relation). For convenience, we supply a fact $\texttt{clause}(c)$ for each clause c, $\texttt{atom}(a)$ for each atom a occurring in some clause, and $\texttt{var}(i, x)$ for each individual or set variable x bound by the ith quantifier. Finally, we still need to declare which quantifiers are existential and which are universal. For

[5] Note that we write X here as a capital letter, as it is customary to do so for set variables. However, in the ASP language we would in fact use a fresh lower-case letter, as we represent the names of both individual and set variables as *constant symbols* in ASP, which may not start with a capital letter.

this, we supply the facts $\texttt{or}(i-1)$ or $\texttt{and}(i-1)$ if the ith quantifier binds a set variable existentially or universally, respectively. If the ith quantifier binds an individual variable x existentially or universally, we supply the rules $\texttt{or}(i-1) \leftarrow \texttt{assign}(x, \_)$ or $\texttt{and}(i-1) \leftarrow \texttt{assign}(x, \_)$, respectively. These rule bodies cause D-FLAT to only map the respective i-tree nodes to "or" or "and" if the corresponding individual variable has been defined. This will allow D-FLAT to delete all subtrees rooted in a node that leaves an individual variable undefined (as required for the final evaluation of the MSO formula via reduction to a Boolean circuit as described in Section 2).

Let t be the current node during a bottom-up traversal of a tree decomposition $\mathcal{T}$ of the input graph $\mathcal{A}$. The i-tree at t shall represent a valid s-tree (cf. Definitions 3 and 4). In particular, an item set of an i-tree node shall encode the label of the respective s-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 of nodes in b. As we assume the matrix of 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 $v \in I_b(X)$; $\texttt{true}(a)$ or $\texttt{false}(a)$, which only occur in leaf item sets, respectively indicate that the atom a is true or false 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 valid s-tree at t. As input, the encoding is provided with a set of rules describing ϕ as well as $\mathcal{T}$ together with the i-trees from the children of t.

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 rules as described above), 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 the s-tree at t after executing the encoding at t. Again, $\mathcal{S}_t$ denotes the local semantic tree at t, while $\mathcal{S}$ is the 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 $\mathcal{S}_t$, which coincides with $\mathcal{S}$ 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 5 and 9) and declare the appropriate item sets (line 28). Additionally, we add the atoms that are true/false under the assignment (cf. rules deriving $\texttt{true}$/$\texttt{false}$) into the leaf item set (lines 29 and 30). 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 in the i-tree of t' (lines 2 and 3) whose assignment is preserved (line 6) and non-deterministically extended (lines 5 and 9). Already determined truth values of atoms are preserved (lines 13 and 14). Again, atoms that become true/false are determined and the appropriate item sets are filled.

```
% Guess a branch for each child i-tree                                    1
extend(0,R) ← root(R).                                                    2
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), sub(R,_).                 3
% Preserve and extend assignment                                          4
{ assign(X,V) : var(_,X) } ← introduced(V).                               5
assign(X,V)   ← extend(_,S), childItem(S,assign(X,V)),                    6
  not removed(V).
assign(X,_nn) ← extend(L,S), childItem(S,assign(X,V)),                    7
  removed(V), iVar(L,X).
% Only join compatible branches; the resulting assignment must be valid   8
← iVar(L,X), assign(X,V;X,W), V ≠ W.                                      9
← extend(L,S;L,T), S ≠ T,                                                 10
  childItem(S,assign(X,_nn);T,assign(X,_nn)).
← extend(L,S;L,T), var(L,X), vertex(V),                                   11
  childItem(S,assign(X,V)), not childItem(T,assign(X,V)).
% Truth values of atoms                                                   12
true (X) ← extend(_,S), childItem(S,true (X)).                            13
false(X) ← extend(_,S), childItem(S,false(X)).                            14
true (edge(X,Y)) ← atom(edge(X,Y)), assign(X,V;Y,W), edge(V,W).           15
false(edge(X,Y)) ← atom(edge(X,Y)), not true(edge(X,Y)),                  16
  assign(X,V;Y,W), not edge(V,W).
true (in(X,Y))   ← atom(in(X,Y)), assign(X,V), assign(Y,V).               17
false(in(X,Y))   ← atom(in(X,Y)), assign(X,V), vertex(V), not             18
  assign(Y,V).
% Truth values of clauses                                                 19
possiblyTrue(C) ← pos(C,A), not false(A).                                 20
possiblyTrue(C) ← neg(C,A), not true (A).                                 21
reject ← clause(C), not possiblyTrue(C).                                  22
trueClause(C) ← pos(C,A), true(A).                                        23
trueClause(C) ← neg(C,A), false(A).                                       24
notAllTrue ← clause(C), not trueClause(C).                                25
accept ← not notAllTrue.                                                  26
% Declare resulting item sets                                            27
item(L,assign(X,V)) ← var(L,X), assign(X,V).                              28
item(L,true (X))    ← length(L), true (X).                                29
item(L,false(X))    ← length(L), false(X).                                30
```

Listing 1.2. MSO model checking with D-FLAT

(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 6), and we set the value of each individual variable that was assigned this vertex to $\star$ (line 7). Truth values of atoms are retained from the predecessor branch, as no additional atoms become true or false according to Lemma 2, The declaration of the item sets proceeds 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 $\star$.

(4) If t is a *join node* with children t_1 and t_2, $\chi(t) = \chi(t_1) = \chi(t_2)$ holds. Here, we guess *a pair* of predecessor branches (lines 2 and 3). 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 Lemma 4, and enforced in lines 10 and 11. 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 6). The sets of true/false atoms under the assignment of b now consists of the union of the true/false atoms in b_1 and the ones in b_2 (lines 13 and 14). In addition, atoms can become false due to line 16. This is in accordance with Lemma 4.

Finally, we show that $\mathcal{A} \models \phi$ holds iff D-FLAT accepts the input. It is not difficult to see that the cases (1) – (4) implement the computation steps that we used in our proofs of Lemmas 1 – 4. The i-tree of any $t \in \mathcal{T}$ can be used to construct $\mathcal{S}_t$ by means of the extension pointers, as can be seen by induction. Furthermore, the atoms evaluating to true/false under the interpretation corresponding to a branch of $\mathcal{S}_t$ are exactly those in the respective leaf item set. If t is the root of $\mathcal{T}$, we obtain $\mathcal{S}$ in this way. D-FLAT's propagation of acceptance statuses according to lines 19 – 26 (cf. Section 3) corresponds to the evaluation of the Boolean circuit obtained from $\mathcal{S}$. It is easy to see that the propagation leads to the same result as the Boolean circuit evaluation in $\mathcal{S}$ because the i-tree at the root of $\mathcal{T}$ is valid. If this propagation finally leads to acceptance, D-FLAT accepts the input; otherwise it rejects.

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.

5 Similar Approaches

Our semantic-tree-based MSO MC approach can be considered as a special case of the game-theoretic approach in [24]. Semantic trees are a classical tool for systematically enumerating truth assignments. Their use goes back to the early days of automated theorem proving [25]. The reason for our modifications of the approach from [24] is that the resulting semantic-tree-based procedure can be implemented in a rather straightforward way via D-FLAT as we have shown in Section 4. In this section, we first discuss the main differences between our semantic-tree-based approach and the more general game-theoretic approach. Then we explain why the latter is not so well suited for implementation via D-FLAT.

The main restriction we impose in this work is that we assume MSO formulas to be in prenex normal form, while [24] requires only negation normal form. In general,

transforming a formula to prenex normal form incurs an increase in the quantifier rank. However, for the purpose of the present work this is acceptable, as we are only interested in an expressibility result here and not in practical efficiency. This restriction allows us to deal with simpler objects during the dynamic programming, which makes the proofs easier and allows for an implementation with D-FLAT, as we will point out in the following.

An extended model checking (EMC) game, as defined in [24], can be represented as a tree where each node represents a position in the game and each edge is a move between positions. EMC games reflect the structure of the formula: Each position corresponds to a subexpression of the formula, and the depth of a node in the game tree is the depth of the respective subformula in the formula tree. Now consider the special case that the formula under consideration is in prenex normal form with quantifier rank n. In this case, each game tree node at depth $d \leq n$ represents a position where the first d variables are interpreted. Analogously, each semantic tree node at depth d represents an interpretation of the first d variables. In this way, EMC games are the game-theoretic counterpart of semantic trees.

Even under the restriction to prenex normal form, there are, however, also notable differences between EMC games and semantic trees: Semantic trees only store variable assignments and, at their leaves, which atoms evaluate to true and which to false, while EMC games reflects the full structure of the formula. In particular, the positions in an EMC game at depth at least n correspond to logical connectives and atomic formulas according to the representation of the matrix as a formula tree. A semantic tree node at depth n, on the other hand, has exactly one child that stores the true as well as the false atoms from the formula. Hence the structure of semantic trees does not depend on the matrix of the formula.

It is easy to see, however, that evaluating a semantic tree by reduction to a Boolean circuit (cf. Section 2) ultimately yields the same result as evaluating the corresponding EMC game: In our reduction to Boolean circuit evaluation we first replaced each leaf of the semantic tree by $\top$ or $\bot$ depending on the truth value of the matrix according to the truth values of the atoms stored in that leaf. Similarly, the EMC game position that represents the same interpretation evaluates to $\top$ if this interpretation satisfies the matrix and to $\bot$ otherwise, as shown in Lemma 1 of [24] (which states that EMC games can easily be converted to MC games, of which it is well known that this property holds). Hence our Boolean circuit evaluation proceeds exactly like the evaluation of EMC games, due to Algorithm 1 of [24].

The considerations so far indicate that semantic trees are essentially just an alternative representation of EMC games that is possible due to our restriction to prenex normal form. In the dynamic programming steps, D-FLAT manipulates (representations of) valid s-trees, the "decomposition-aware" variant of semantic trees, as described in Section 4. It is here that the motivation for our simplifications becomes evident.

Valid s-trees again have EMC games as their game-theoretic counterpart. However, in an s-tree, individual variables can be assigned the special value $\star$, which stands for any forgotten vertex. In contrast, EMC games do not use a special value in place of forgotten vertices. Instead, when a vertex is forgotten, at first it remains in the EMC game that is being manipulated in the dynamic programming of [24]. However, the resulting

EMC game is subsequently reduced via Algorithm 3 in [24]. In this reduction, for any two "equivalent" subgames reachable from any position, only one of these subgames is retained. Two subgames are equivalent (cf. Definition 5 in [24]) if there is a bijection between their positions such that each pair of corresponding positions is "equivalent", and two positions are equivalent if, informally speaking, for each bag element the variables being assigned this element are the same in both positions. Note that this notion of equivalence only considers bag elements but does not concern forgotten vertices. Therefore, if two subgames only differ with regard to forgotten vertices, they are still equivalent and Algorithm 3 in [24] removes one of them.

Hence the dynamic programming for EMC games ensures that the sizes of the games that are being computed stays bounded by some function that depends only on the treewidth and the formula size (cf. Lemma 9 in [24]). This reduction is based on an isomorphism criterion that only takes elements of the current bag into account. As D-FLAT is constructed in such a way that it only removes redundancies if two subtrees are isomorphic and the corresponding item sets are equal (cf. Section 4), it does not disregard information about forgotten vertices. Forgetting information is the duty of the user-specified ASP encoding (in our case Listing 1.2). If, for instance, an i-tree node has two children N_1, N_2 corresponding to the same individual variable z such that N_1 assigns a forgotten vertex to z and N_2 assigns a different forgotten vertex to z, then D-FLAT will never discard N_1 or N_2. This is why in our approach we set the value of an individual variable to $\star$ if this variable has been assigned a vertex that does not appear in the current bag. In this way, the equality-based procedure for removing redundancies in D-FLAT can perform a task analogous to the reduction procedure for EMC games.

A related way to eliminate redundancies by means of an equality test instead of an isomorphism test was presented in [28], which deals with an FPT algorithm for MSO MC on graphs of bounded rankwidth. That paper introduces *characteristic trees*, whose structure is very much like semantic trees. The nodes along a path from a characteristic tree's root to one of its leaves also represent assignments to the variables in the same order as the variables occur in the formula. Unlike the approaches based on semantic trees or EMC games, the characteristic-tree-based approach does not perform partial evaluation of the formula (i.e., discarding a subtree of an s-tree or a subgame of an EMC game as soon as the truth value of the corresponding subformula can be established). Another difference between characteristic trees and semantic trees lies in the fact that characteristic trees store not the actual value of a variable once it is assigned but integers that represent equivalence classes over the assigned variables. This is done in order to keep the sizes of the characteristic trees bounded by a function of the rankwidth. The usage of integers that represent equivalence classes plays a similar role to the usage of $\star$ in this paper. Again, our semantic-tree-based approach is simpler at the price of being less general. However, as we have explained above, for the purpose of the current paper our relatively restricted notions are sufficient.

6 Conclusion

There is vivid interest in turning theoretical tractability results obtained via Courcelle's Theorem into concrete computation that is feasible in practice [27]. 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 [4] suggest that D-FLAT is a promising means to do so. In contrast to recent MSO-based systems [24, 26] where the problem is expressed in a monolithic way, D-FLAT allows the user to define the dynamic programming algorithm on a tree decomposition via ASP. Like in the Datalog approach [21], this admits a declarative specification while still being able to take advantage of domain knowledge. However, the approach in [21] 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.

The declarative language provided by D-FLAT leads to implementations of algorithms that leverage bounded treewidth in a natural way, as the examples in Section 3 and [2, 1, 4] 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.

Our approach via s-trees follows the ideas of the approaches based on extended MC games in [24] and on characteristic trees in [28]. We adapted these concepts by introducing s-trees in order to provide a data structure that lends itself to be used in conjunction with D-FLAT. Compared to solvers like [24, 26], which require just a formulization of the problem in MSO, D-FLAT has the advantage that the user can perform optimizations by incorporating problem-specific knowledge while still remaining on a rather high level of declarativity. Indeed, several such optimizations would be possible for our implementation of MSO MC described in this paper – however, we aimed only at providing an expressibility result rather than implementing an efficient competitor to dedicated MSO solvers.

Future work in particular includes a comparison of the ASP-based D-FLAT approach with the LISP-based *Autograph* approach [12] regarding both the range of theoretical applicability and practical efficiency. *Autograph* allows to specify the problem at hand as a formula, which it then translates to combinations of (pre-defined) fly-automata. Furthermore, we plan to compare the performance of D-FLAT to the *Sequoia* system, which implements the approach in [24, 26].

Acknowledgments

This work is supported by the Austrian Science Fund (FWF) projects P25518, P25607 and Y698.

References

1. Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., Woltran, S.: D-FLAT: Progress report. Tech. Rep. DBAI-TR-2014-86, Vienna University of Technology (2014)
2. Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., Woltran, S.: The D-FLAT system for dynamic programming on tree decompositions. In: Proc. JELIA. pp. 558–572 (2014)
3. Alviano, M., Calimeri, F., Faber, W., Ianni, G., Leone, N.: Function symbols in ASP: Overview and perspectives. In: Nonmonotonic Reasoning – Essays Celebrating Its 30th Anniversary, pp. 1–24. College Publications, London (2011)
4. Bliem, B., Morak, M., Woltran, S.: D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. TPLP 12(4-5), 445–464 (2012)
5. Bliem, B., Pichler, R., Woltran, S.: Declarative dynamic programming as an alternative realization of Courcelle's theorem. In: Proc. IPEC. pp. 28–40 (2013)
6. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybern. 11(1-2), 1–22 (1993)
7. Bodlaender, H.L.: Discovering treewidth. In: Proc. SOFSEM. LNCS, vol. 3381, pp. 1–16. Springer (2005)
8. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
9. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Proc. ICLP. LNCS, vol. 5366, pp. 407–424. Springer (2008)
10. Chandra, A.K., Kozen, D., Stockmeyer, L.J.: Alternation. J. ACM 28(1), 114–133 (1981)
11. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Inf. Comput. 85(1), 12–75 (1990)
12. Courcelle, B., Durand, I.: Computations by fly-automata beyond monadic second-order logic. CoRR abs/1305.7120 (2013)
13. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science, Springer (1999)
14. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. J. ACM 49(6), 716–752 (2002)
15. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science, Springer (2006)
16. Frick, M., Grohe, M.: The complexity of first-order and monadic second-order logic revisited. Ann. Pure Appl. Logic 130(1-3), 3–31 (2004)
17. 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)
18. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: The potsdam answer set solving collection. AI Commun. 24(2), 107–124 (2011)
19. Gelfond, M., Leone, N.: Logic programming and knowledge representation – the A-Prolog perspective. Artif. Intell. 138(1-2), 3–38 (2002)
20. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365–386 (1991)

21. Gottlob, G., Pichler, R., Wei, F.: Monadic datalog over finite structures of bounded treewidth. ACM Trans. Comput. Log. 12(1) (2010)
22. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. 13(4), 571–586 (2002)
23. Kloks, T.: Treewidth: Computations and Approximations, LNCS, vol. 842. Springer (1994)
24. Kneis, J., Langer, A., Rossmanith, P.: Courcelle's theorem – a game-theoretic approach. Discrete Optimization 8(4), 568–594 (2011)
25. Kowalski, R.A., Hayes, P.J.: Semantic trees in automated theorem proving. In: Machine Intelligence, vol. 4, pp. 87–101. Edinburgh University Press (1969)
26. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: Proc. ALENEX. pp. 55–63. SIAM / Omnipress (2012)
27. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Practical algorithms for MSO model-checking on tree-decomposable graphs. Computer Science Review 13-14, 39–74 (2014)
28. 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. LNCS, vol. 6648, pp. 505–516. Springer (2011)
29. Lifschitz, V.: What is answer set programming? In: Proc. AAAI. pp. 1594–1597. AAAI Press (2008)
30. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: A 25-Year Perspective, pp. 375–398. Springer (1999)
31. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics And Its Applications, Oxford University Press (2006)
32. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. 25(3-4), 241–273 (1999)
33. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)