Millstream Systems – a Formal Model for Linking Language Modules by Interfaces

Suna Bensch
Department of Computing Science,
Umeå University, Umeå, Sweden
suna@cs.umu.se

Frank Drewes
Department of Computing Science,
Umeå University, Umeå, Sweden
drewes@cs.umu.se

Abstract

We introduce Millstream systems, a formal model consisting of modules and an interface, where the modules formalise different aspects of language, and the interface links these aspects with each other.

1 Credits

This work is partially supported by the project *Tree Automata in Computational Language Technology* within the Sweden – South Africa Research Links Programme. A preliminary but more detailed version of this article is available as a technical report (Bensch and Drewes, 2009).

2 Introduction

Modern linguistic theories (Sadock, 1991; Jackendoff, 2002) promote the view that different aspects of language, such as phonology, morphology, syntax, and semantics should be viewed as autonomous modules that work simultaneously and are linked with each other by interfaces that describe their interaction and interdependency. Formalisms in modern computational linguistics which establish interfaces between different aspects of language are the Combinatory Categorical Grammar (CCG), the Functional Generative Description (FGD), the Head-Driven Phrase Structure Grammar (HPSG), the Lexical Functional Grammar (LFG), and the Extensible Dependency Grammar (XDG).[1] Here, we propose Millstream systems, an approach from a formal language theoretic point of view which is based on the same ideas as XDG, but uses tree-generating modules of arbitrary kinds.

Let us explain in slightly more detail what a Millstream system looks like. A Millstream system contains any number of tree generators, called its modules. Such a tree generator is any device that specifies a tree language. For example, a tree generator may be a context-free grammar, tree adjoining grammar, a finite-state tree automaton, a dependency grammar, a corpus, human input, etc. Even within a single Millstream system, the modules need not be of the same kind, since they are treated as "black boxes". The Millstream system links the trees generated by the modules by an interface consisting of logical formulas.

Suppose that a Millstream system has k modules. Then the interface consists of interface rules in the form of logical expressions that establish links between the (nodes of the) trees $t_1, \ldots, t_k$ that are generated by the individual modules. Thus, a valid combination of trees is not just any collection of trees $t_1, \ldots, t_k$ generated by the k modules. It also includes, between these structures, interconnecting links that represent their relationships and that must follow the rules expressed by the interface. Grammaticality, in terms of a Millstream system, means that the individual structures must be valid (i.e., generated by the modules) and are linked in such a way that all interface rules are logically satisfied. A Millstream system can thus be considered to perform independent concurrent derivations of autonomous modules, enriched by an interface that establishes links between the outputs of the modules, thus constraining the acceptable configurations.

Millstream systems may, for example, be of interest for natural language understanding and natural language generation. Simply put, the task of natural language understanding is to construct a suitable semantic representation of a sentence that has been heard (phonology) and parsed (syntax). Within the framework of Millstream systems this corresponds to the problem where we are given a syntactic tree (and possibly a phonological tree if such a module is involved) and the goal is to construct an appropriate semantic tree. Con-

[1] See, e.g., (Dalrymple, 2001; Sgall et al., 1986; Pollard and Sag, 1994; Steedman, 2000; Debusmann, 2006; Debusmann and Smolka, 2006).

Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing, ACL 2010, pages 28–36,
Uppsala, Sweden, 16 July 2010. ©2010 Association for Computational Linguistics

versely, natural language generation can be seen as the problem to construct an appropriate syntactic (and/or phonological) tree from a given semantic tree. In abstract terms, the situations just described are identical. We refer to the problem as the *completion problem*. While the current paper is mainly devoted to the introduction and motivation of Millstream systems, in (Bensch et al., 2010) the completion problem is investigated for so-called regular MSO Millstream systems, i.e. systems in which the modules are regular tree grammars (or, equivalently, finite tree automata) and the interface conditions are expressed in monadic second-order (MSO) logic. In Section 7, the results obtained so far are briefly summarised.

Now, let us roughly compare Millstream systems with XDG. Conceptually, the k modules of a Millstream system correspond to the k *dimensions* of an XDG. In an XDG, a configuration consists of dependency structures $t_1, \ldots, t_k$. The interface of a Millstream system corresponds to the *principles* of the XDG. The latter are logical formulas that express conditions that the collection of dependency structures must fulfill.

The major difference between the two formalisms lies in the fact that XDG inherently builds upon dependency structures, whereas the modules of a Millstream system are arbitrary tree generators. In XDG, each of $t_1, \ldots, t_k$ is a dependency analysis of the sentence considered. In particular, they share the yield and the set of nodes (as the nodes of a dependency tree correspond to the words in the sentence analysed, and its yield is that sentence). Millstream systems do not make similar assumptions, which means that they may give rise to new questions and possibilities:

- The purpose of a Millstream system is not necessarily the analysis of sentences. For example, a Millstream system with two modules could translate one language into another. For this, tree grammars representing the source and target languages could be used as modules, with an interface expressing that t_2 is a correct translation of t_1. This scenario makes no sense in the context of XDG, because the sentences represented by t_1 and t_2 differ. Many similar applications of Millstream system may the thought of, for example correction or simplification of sentences.

- As the modules may be arbitrary devices specifying tree languages, they contribute their own generative power and theoretical properties to the whole (in contrast to XDG, which does not have such a separation). This makes it possible to apply known results from tree language theory, and to study the interplay between different kinds of modules and interface logics.

- The fact that the individual modules of a Millstream system may belong to different classes of tree generators could be linguistically valuable. For example, a Millstream system combining a dependency grammar module with a regular tree grammar module, could be able to formalise aspects of a given natural language that cannot be formalised by using only one of these formalisms.

- For Millstream systems whose modules are generative grammar formalisms (such as regular tree grammars, tree-adjoining grammars and context-free tree grammars), it will be interesting to study conditions under which the Millstream system as a whole becomes generative, in the sense that well-formed configurations can be constructed in a step-by-step manner based on the derivation relations of the individual modules.

Let us finally mention another, somewhat subtle difference between XDG and Millstream systems. In XDG, the interfaces are dimensions on their own. For example, an XDG capturing the English syntax and semantics would have three dimensions, namely syntax, semantics, and the syntax-semantics interface. An analysis of a sentence would thus consist of three dependency trees, where the third one represents the relation between the other two. In contrast, a corresponding Millstream system would only have two modules. The interface between them is considered to be conceptually different and establishes direct links between the trees that are generated by the two modules. One of our tasks (which is, however, outside the scope of this contribution) is a study of the formal relation between XDG and Millstream systems, to achieve a proper understanding of their similarities and differences.

The rest of the paper is organised as follows. In the next section, we discuss an example illustrating the linguistic notions and ideas that Millstream systems attempt to provide a formal basis for. After some mathematical preliminaries, which

are collected in Section 4, the formal definition of Millstream systems is presented in Section 5. Section 6 contains examples and remarks related to Formal Language Theory. Finally, Section 7 discusses preliminary results and future work.

3 Linguistical Background

In this section, we discuss an example, roughly following (Jackendoff, 2002), that illustrates the linguistic ideas that have motivated our approach. Figure 1 shows the phonological, syntactical and semantical structure, depicted as trees (a), (b) and (c), respectively of the sentence *Mary likes Peter*. Trees are defined formally in the next section, for the time being we assume the reader to be familiar with the general notion of a tree as used in linguistics and computer science.

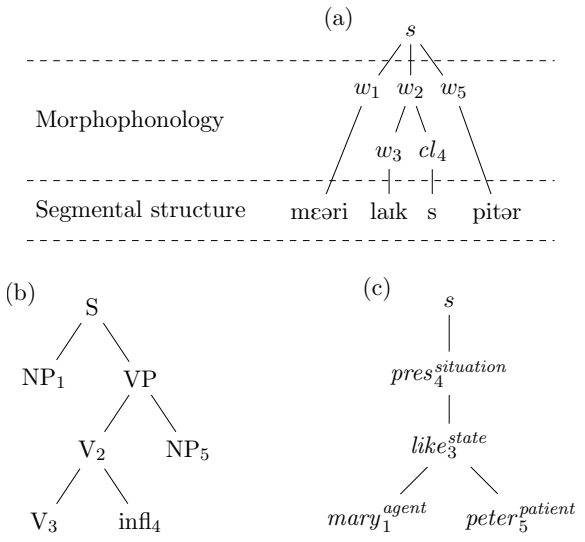

Figure 1: Phonological, syntactical and semantical structure of *Mary likes Peter*.

The segmental structure in the phonological tree (a) is the basic pronunciation of the sentence *Mary likes Peter*, where each symbol represents a speech sound. This string of speech sound symbols is structured into phonological words by morphophonolgy. The morphophonological structure in our example consists of the three full phonological words mɛəri, laɪk, pitər and of the clitic *s*. The clitic is attached to the adjacent phonological word, thus forming a larger phonological constituent. The syntactical tree (b) depicts the syntactical constituents. The sentence S is divided into a noun phrase NP and a verb phrase VP. The verb phrase VP is divided into an inflected verb V and a noun phrase NP. The inflected verb consists of its uninflected form and its inflection, which refers, in our example, to the grammatical features present tense and third person singular. The semantical tree (c) depicts the semantical constituents. In our example, $like$ is a function of type $state$ and takes two arguments, namely $mary$ and $peter$ which are of type $agent$ and $patient$.

The structure of *Mary likes Peter* is not just the sum of its phonological, syntactical and semantical structures. It also includes the relationships between certain constituents in these tree structures. To illustrate these relationships we use indices in Figure 1. The sole role of the indices here is to express the linguistic relationships among coindexed constituents. The indices do not occur in the formalisation, where they are replaced by logical links relating the nodes that, in the figure, carry the same indices.[2] The morphophonological word w_1, for example, is linked with the noun phrase NP_1 in the syntactical tree and with the conceptual constituent $mary_1^{agent}$ in the semantical tree. This illustrates that w_1, NP_1, and $mary_1^{agent}$ are the corresponding morphophonological, syntactical and semantical representations of *Mary*, respectively. But there are also correspondences that concern only the phonological and syntactical trees, excluding the semantical tree. For example, the inflected word V_2 in the syntactical structure corresponds to the phonological word w_2, but has no link to the semantical structure whatsoever.

4 Preliminaries

The set of non-negative integers is denoted by $\mathbb{N}$, and $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. For $k \in \mathbb{N}$, we let $[k] = \{1, \ldots, k\}$. For a set S, the set of all nonempty finite sequences (or strings) over S is denoted by S^+; if the empty sequence ϵ is included, we write S^*. As usual, $A_1 \times \cdots \times A_k$ denotes the Cartesian product of sets $A_1, \ldots, A_k$. The transitive and reflexive closure of a binary relation $\Rightarrow \subseteq A \times A$ on a set A is denoted by $\Rightarrow^*$. A *ranked alphabet* is a finite set Σ of pairs (f, k), where f is a symbol and $k \in \mathbb{N}$ is its *rank*. We denote (f, k) by $f^{(k)}$, or simply by f if k is understood or of lesser importance. Further, we let $\Sigma^{(k)} = \{f^{(n)} \in \Sigma \mid n = k\}$. We define trees over Σ in one of the standard ways, by identifying the nodes of a tree t with sequences of natural numbers. Intuitively, such a sequence

[2] The reader is referred to (Bensch and Drewes, 2009) for the proper formalisation of the example in terms of a Millstream system.

shows that path from the root of the tree to the node in question. In particular, the root is the empty sequence ϵ.

Formally, the set T_Σ of trees over Σ consists of all mappings $t\colon V(t) \to \Sigma$ (called trees) with the following properties:

- The set $V(t)$ of *nodes of* t is a finite and non-empty prefix-closed subset of $\mathbb{N}_+^*$. Thus, for every node $vi \in V(t)$ (where $i \in \mathbb{N}_+$), its *parent* v is in $V(t)$ as well.
- For every node $v \in V(t)$, if $t(v) = f^{(k)}$, then $\{i \in \mathbb{N} \mid vi \in V(t)\} = [k]$. In other words, the *children* of v are $v1, \ldots, vk$.

Let $t \in \mathrm{T}_\Sigma$ be a tree. The *root* of t is the node ϵ. For every node $v \in V(t)$, the *subtree of* t *rooted at* v is denoted by t/v. It is defined by $V(t/v) = \{v' \in \mathbb{N}^* \mid vv' \in V(t)\}$ and, for all $v' \in V(t/v)$, $(t/v)(v') = t(vv')$. We shall denote a tree t as $f[t_1, \ldots, t_k]$ if $t(\epsilon) = f^{(k)}$ and $t/i = t_i$ for $i \in [k]$. In the special case where $k = 0$ (i.e., $V(t) = \{\epsilon\}$), the brackets may be omitted, thus denoting t as f. For a set S of trees, the set of all trees of the form $f[t_1, \ldots, t_k]$ such that $f^{(k)} \in \Sigma$ and $t_1, \ldots, t_k \in S$ is denoted by $\Sigma(S)$. For a tuple $T \in \mathrm{T}_\Sigma^k$, we let $V(T)$ denote the set $\{(i, v) \mid i \in [k] \text{ and } v \in V(t_i)\}$. Thus, $V(T)$ is the disjoint union of the sets $V(t_i)$. Furthermore, we let $V(T, i)$ denote the ith component of this disjoint union, i.e., $V(T, i) = \{i\} \times V(t_i)$ for all $i \in [k]$. A *tree language* is a subset of T_Σ, for a ranked alphabet Σ, and a Σ*-tree generator* (or simply *tree generator*) is any sort of formal device G that determines a tree language $L(G) \subseteq \mathrm{T}_\Sigma$. A typical sort of tree generator, which we will use in our examples, is the regular tree grammar.

Definition 1 (regular tree grammar)**.** A *regular tree grammar* is a tuple $G = (N, \Sigma, R, S)$ consisting of disjoint ranked alphabets N and Σ of *nonterminals* and *terminals*, where $N = N^{(0)}$, a finite set R of rules $A \to r$, where $A \in N$ and $r \in \mathrm{T}_{\Sigma \cup N}$, and an *initial nonterminal* $S \in N$.

Given trees $t, t' \in \mathrm{T}_{\Sigma \cup N}$, there is a *derivation step* $t \Rightarrow t'$ if t' is obtained from t by replacing a single occurrence of a nonterminal A with r, where $A \to r$ is a rule in R. The *regular tree language generated by* G is

$$L(G) = \{t \in \mathrm{T}_\Sigma \mid S \overset{*}{\Rightarrow} t\}.$$

It is well known that a string language L is context-free if and only if there is a regular tree language L', such that $L = \mathrm{yield}(L')$. Here, $\mathrm{yield}(L') = \{\mathrm{yield}(t) \mid t \in L'\}$ denotes the set of all yields of trees in L', the yield $\mathrm{yield}(t)$ of a tree t being the string obtained by reading its leaves from left to right.

5 Millstream Systems

Throughout the rest of this paper, let Λ denote any type of predicate logic that allows us to make use of n-ary predicates symbols. We indicate the arity of predicate symbols in the same way as the rank of symbols in ranked alphabets, i.e., by writing $P^{(n)}$ if P is a predicate symbol of arity n. The set of all well-formed formulas in Λ without free variables (i.e., the set of sentences of Λ) is denoted by F_Λ. If S is a set, we say that a predicate symbol $P^{(n)}$ is S-typed if it comes with an associated type $(s_1, \ldots, s_n) \in S^n$. We write $P\colon s_1 \times \cdots \times s_n$ to specify the type of P. Recall that an n-ary predicate ψ on D is a function $\psi\colon D^n \to \{\mathrm{true}, \mathrm{false}\}$. Alternatively, ψ can be viewed as a subset of D^n, namely the set of all $(d_1, \ldots, d_n) \in D^n$ such that $\psi(d_1, \ldots, d_n) = \mathrm{true}$. We use these views interchangeably, selecting whichever is more convenient. Given a (finite) set $\mathcal{P}$ of predicate symbols, a logical structure $\langle D; (\psi_P)_{P \in \mathcal{P}} \rangle$ consists of a set D called the domain and, for each $P^{(n)} \in \mathcal{P}$, a predicate $\psi_P \subseteq D^n$. If an existing structure Z is enriched with additional predicates $(\psi_P)_{P \in \mathcal{P}'}$ (where $\mathcal{P} \cap \mathcal{P}' = \emptyset$), we denote the resulting structure by $\langle Z; (\psi_P)_{P \in \mathcal{P}'} \rangle$. In this paper, we will only consider structures with finite domains. To represent (tuples of) trees as logical structures, consider a ranked alphabet Σ, and let r be the maximum rank of symbols in Σ. A tuple $T = (t_1, \ldots, t_k) \in \mathrm{T}_\Sigma^k$ will be represented by the structure

$$|T| = \langle V(T); (V_i)_{i \in [k]}, (\mathrm{lab}_g)_{g \in \Sigma}, (\downarrow_i)_{i \in [r]} \rangle$$

consisting of the domain $V(T)$ and the predicates $V_i^{(1)}$ ($i \in [k]$), $\mathrm{lab}_g^{(1)}$ ($g \in \Sigma$) and $\downarrow_i^{(2)}$ ($i \in [r]$). The predicates are given as follows:

- For every $i \in [k]$, $V_i = V(T, i)$. Thus, $V_i(d)$ expresses that d is a node in t_i (or, to be precise, that d represents a node of t_i in the disjoint union $V(T)$).
- For every $g \in \Sigma$, $\mathrm{lab}_g = \{(i, v) \in V(T) \mid i \in [k] \text{ and } t_i(v) = g\}$. Thus, $\mathrm{lab}_g(d)$ expresses that the label of d is g.
- For every $j \in [r]$, $\downarrow_j = \{((i, v), (i, vj)) \mid i \in [k] \text{ and } v, vj \in V(t_i)\}$. Thus, $\downarrow_j(d, d')$

expresses that d' is the jth child of d in one of the trees $t_1, \ldots, t_k$. In the following, we write $d \downarrow_j d'$ instead of $\downarrow_j(d, d')$.

Note that, in the definition of $|T|$, we have blurred the distinction between predicate symbols and their interpretation as predicates, because this interpretation is fixed. Especially in intuitive explanations, we shall sometimes also identify the logical structure $|T|$ with the tuple T it represents.

To define Millstream systems, we start by formalising our notion of interfaces. The idea is that a tuple $T = (t_1, \ldots, t_k)$ of trees, represented by the structure $|T|$, is augmented with additional *interface links* that are subject to logical conditions. An interface may contain finitely many different kinds of interface links. Formally, the collection of all interface links of a given kind is viewed as a logical predicate. The names of the predicates are called interface symbols. Each interface symbol is given a type that indicates which trees it is intended to link with each other.

For example, if we want to make use of ternary links called TIE, each linking a node of t_1 with a node of t_3 and a node of t_4, we use the interface symbol TIE$\colon 1 \times 3 \times 4$. This interface symbol would then be interpreted as a predicate $\psi_{\text{TIE}} \subseteq V(T,1) \times V(T,3) \times V(T,4)$. Each triple in ψ_{TIE} would thus be an interface link of type TIE that links a node in $V(t_1)$ with a node in $V(t_3)$ and a node in $V(t_4)$.

Definition 2 (interface)**.** Let Σ be a ranked alphabet. An *interface on* T^k_Σ ($k \in \mathbb{N}$) is a pair $INT = (\mathcal{I}, \Phi)$, such that

- $\mathcal{I}$ is a finite set of $[k]$-typed predicate symbols called *interface symbols*, and
- Φ is a finite set of formulas in F_Λ that may, in addition to the fixed vocabulary of Λ, contain the predicate symbols in $\mathcal{I}$ and those of the structures $|T|$ (where $T \in \mathrm{T}^k_\Sigma$). These formulas are called *interface conditions*.

A *configuration* (w.r.t. INT) is a structure $C = \langle |T|; (\psi_I)_{I \in \mathcal{I}} \rangle$, such that

- $T \subseteq \mathrm{T}^k_\Sigma$,
- $\psi_I \subseteq V(T, i_1) \times \cdots \times V(T, i_l)$ for every interface symbol $I\colon i_1 \times \cdots \times i_l$ in $\mathcal{I}$, and
- C satisfies the interface conditions in Φ (if each symbol $I \in \mathcal{I}$ is interpreted as ψ_I).

Note that several interfaces can always be combined into one by just taking the union of their sets of interface symbols and interface conditions.

Definition 3 (Millstream system)**.** Let Σ be a ranked alphabet and $k \in \mathbb{N}$. A *Millstream system* (MS, for short) is a system of the form $MS = (M_1, \ldots, M_k; INT)$ consisting of Σ-tree generators $M_1, \ldots, M_k$, called the *modules* of MS, and an interface INT on T^k_Σ. The language $L(MS)$ generated by MS is the set of all configurations $\langle |T|; (\psi_I)_{I \in \mathcal{I}} \rangle$ such that $T \in L(M_1) \times \cdots \times L(M_k)$.

Sometimes we consider only some of the trees in these tuples. For this, if MS is as above and $1 \leq i_1 < \cdots < i_l \leq k$, we define the notation

$$L^{M_{i_1} \times \cdots \times M_{i_l}}(MS) = \{(t_{i_1}, \ldots, t_{i_l}) \mid \langle |(t_1, \ldots, t_k); (\psi_I)_{I \in \mathcal{I}}| \rangle \in L(MS)\}.$$

The reader should note that, intentionally, Millstream systems are not a priori "generative". Even less so, they are "derivational" by nature. This is because there is no predefined notion of derivation that allows us to create configurations by means of a stepwise (though typically nondeterministic) procedure. In fact, there cannot be one, unless we make specific assumptions regarding the way in which the modules work, but also regarding the logic Λ and the form of the interface conditions that may be used. Similarly, as mentioned in the introduction, there is no predefined order of importance or priority among the modules.

6 Examples and Remarks Related to Formal Language Theory

The purpose of this section is to indicate, by means of examples and easy observations, that Millstream systems are not only linguistically well motivated, but also worth studying from the point of view of computer science, most notably regarding their algorithmic and language-theoretic properties. While this kind of study is beyond the scope of the current article, part of our future research on Millstream systems will be devoted to such questions.

Example 1. Let Λ be ordinary first-order logic with equality, and consider the Millstream system MS over $\Sigma = \{\circ^{(2)}, a^{(0)}, b^{(0)}, c^{(0)}, d^{(0)}\}$ which consists of two identical modules $M_1 = M_2$ that simply generate T_Σ (e.g., using the regular tree grammar with the single nonterminal S and the

rules[3] $S \to \circ[S,S] \mid a \mid b \mid c \mid d$) and a single interface symbol BIJ: 1×2 with the interface conditions

$$\begin{aligned}
&\forall x\colon \mathrm{lab}_{\{a,b,c,d\}}(x) \leftrightarrow \\
&\qquad \exists y\colon \mathrm{BIJ}(x,y) \vee \mathrm{BIJ}(y,x), \\
&\forall x,y,z\colon (\mathrm{BIJ}(x,y) \wedge \mathrm{BIJ}(x,z) \vee \\
&\qquad \mathrm{BIJ}(y,x) \wedge \mathrm{BIJ}(z,x)) \to y = z, \\
&\forall x,y\colon \mathrm{BIJ}(x,y) \to \\
&\qquad \bigvee_{z \in \{a,b,c,d\}} (\mathrm{lab}_z(x) \wedge \mathrm{lab}_z(y)).
\end{aligned}$$

The first interface condition expresses that all and only the leaves of both trees are linked. The second expresses that no leaf is linked with two or more leaves. In effect, this amounts to saying that BIJ is a bijection between the leaves of the two trees. The third interface condition expresses that this bijection is label preserving. Altogether, this amounts to saying that the yields of the two trees are permutations of each other; see Figure 2.

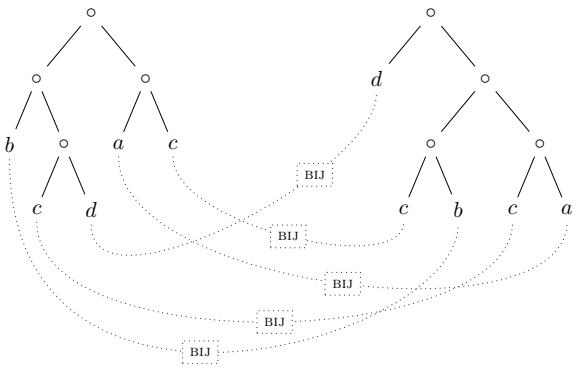

Figure 2: An element of $L(MS)$ in Example 1.

Now, let us replace the modules by slightly more interesting ones. For a string w over $\{A, B, a, b, c, d\}$, let $\underline{w}$ denote any tree over $\{\circ^{(2)}, A^{(0)}, B^{(0)}, a^{(0)}, b^{(0)}, c^{(0)}, d^{(0)}\}$ with $\mathrm{yield}(\underline{w}) = w$. (For example, we may choose $\underline{w}$ to be the left comb whose leaf symbols are given by w.) Let the Millstream system MS' be defined as MS, but using the modules $M_1' = (\{A, B, C, D\}, \Sigma, R_1, A)$ and $M_2' = (\{A, B\}, \Sigma, R_2, A)$ with the following rules:

$$\begin{aligned}
R_1' = \{&A \to \underline{aA} \mid \underline{aB},\ B \to \underline{bB} \mid \underline{bC}, \\
&C \to \underline{cC} \mid \underline{cD},\ D \to \underline{dD} \mid \underline{d}\}, \\
R_2' = \{&A \to \underline{acA} \mid \underline{acB},\ B \to \underline{bdB} \mid \underline{bd}\}.
\end{aligned}$$

Thus, M_1' and M_2' are the "standard" grammars (written as regular tree grammars) that generate the regular languages $\{a^k b^l c^m d^n \mid k,l,m,n \geq 1\}$ and $\{(ac)^m (bd)^n \mid m,n \geq 1\}$. The interface makes sure that $L^{M_1' \times M_2'}(MS')$ contains only those pairs of trees t_1, t_2 in which $\mathrm{yield}(t_1)$ is a permutation of $\mathrm{yield}(t_2)$. As a consequence, it follows that $\mathrm{yield}(L^{M_1'}(MS)) = \{a^m b^n c^m d^n \mid m,n \geq 1\}$.

The next example discusses how top-down tree transductions can be implemented as Millstream systems.

Example 2 (top-down tree transduction)**.** Recall that a *tree transduction* is a binary relation $\tau \subseteq \mathrm{T}_\Sigma \times \mathrm{T}_{\Sigma'}$, where Σ and Σ' are ranked alphabets. The set of trees that a tree $t \in \mathrm{T}_\Sigma$ is transformed into is given by $\tau(t) = \{t' \in \mathrm{T}_{\Sigma'} \mid (t,t') \in \tau\}$. Obviously, every Millstream system of the form $MS = (M_1, M_2; INT)$ defines a tree transduction, namely $L^{M_1 \times M_2}(MS)$. Let us consider a very simple instance of a deterministic top-down tree transduction τ (see, e.g., (Gécseg and Steinby, 1997; Fülöp and Vogler, 1998; Comon et al., 2007) for definitions and references regarding top-down tree transductions), where $\Sigma = \Sigma' = \{f^{(2)}, g^{(2)}, a^{(0)}\}$. We transform a tree $t \in \mathrm{T}_\Sigma$ into the tree obtained from t by interchanging the subtrees of all top-most fs (i.e., of all nodes that are labelled with f and do not have an ancestor that is labelled with f as well) and turning the f at hand into a g. To accomplish this, a top-down tree transducer would use two states, say SWAP and COPY, to traverse the input tree from the top down, starting in state SWAP. Whenever an f is reached in this state, its subtrees are interchanged and the traversal continues in parallel on each of the subtrees in state COPY. The only purpose of this state is to copy the input to the output without changing it. Formally, this would be expressed by the following term rewrite rules, viewing the states as symbols of rank 1:

$$\begin{aligned}
\mathrm{SWAP}[f[x_1,x_2]] &\to g[\mathrm{COPY}[x_2], \mathrm{COPY}[x_1]], \\
\mathrm{COPY}[f[x_1,x_2]] &\to f[\mathrm{COPY}[x_1], \mathrm{COPY}[x_2]], \\
\mathrm{SWAP}[g[x_1,x_2]] &\to g[\mathrm{SWAP}[x_1], \mathrm{SWAP}[x_2]], \\
\mathrm{COPY}[g[x_1,x_2]] &\to g[\mathrm{COPY}[x_1], \mathrm{COPY}[x_2]], \\
\mathrm{SWAP}[a] &\to a, \\
\mathrm{COPY}[a] &\to a.
\end{aligned}$$

(We hope that these rules are intuitive enough to be understood even by readers who are unfamiliar with top-down tree transducers, as giving the formal definition of top-down tree transducers would be out of the scope of this article.) We mimic the behaviour of the top-down tree transducer us-

[3] As usual, $A \to r \mid r'$ stands for $A \to r,\ A \to r'$.

ing a Millstream system with interface symbols $\text{SWAP}\colon 1 \times 2$ and $\text{COPY}\colon 1 \times 2$. Since the modules simply generate T_Σ, they are not explicitly discussed. The idea behind the interface is that an interface link labelled $q \in \{\text{SWAP}, \text{COPY}\}$ links a node v in the input tree with a node v' in the output tree if the simulated computation of the tree transducer reaches v in state q, resulting in node v' in the output tree. First, we specify that the initial state is SWAP, which simply means that the roots of the two trees are linked by a SWAP link:

$$\forall x, y\colon root_1(x) \wedge root_2(y) \rightarrow \text{SWAP}(x, y),$$

where $root_i$ is defined as $root_i(x) \equiv V_i(x) \wedge \nexists y\colon y \downarrow_1 x$. It expresses that x is the root of tree i. The next interface condition corresponds to the first rule of the simulated top-down tree transducer:

$$\forall x, y, x_1, x_2\colon \text{SWAP}(x, y) \wedge \text{lab}_f(x) \wedge x \downarrow_1 x_1 \wedge x \downarrow_2 x_2 \rightarrow \text{lab}_g(y) \wedge \exists y_1, y_2\colon y \downarrow_1 y_1 \wedge y \downarrow_2 y_2 \wedge \text{COPY}(x_1, y_2) \wedge \text{COPY}(x_2, y_1).$$

In a similar way, the remaining rules are turned into interface conditions, e.g.,

$$\forall x, y, x_1, x_2\colon \text{COPY}(x, y) \wedge \text{lab}_f(x) \wedge x \downarrow_1 x_1 \wedge x \downarrow_2 x_2 \rightarrow \text{lab}_f(y) \wedge \exists y_1, y_2\colon y \downarrow_1 y_1 \wedge y \downarrow_2 y_2 \wedge \text{COPY}(x_1, y_1) \wedge \text{COPY}(x_2, y_2).$$

The reader should easily be able to figure out the remaining interface conditions required.

One of the elements of $L(MS)$ is shown in Figure 3. It should not be difficult to see that, indeed, $L^{M_1 \times M_2}(MS) = \tau$.

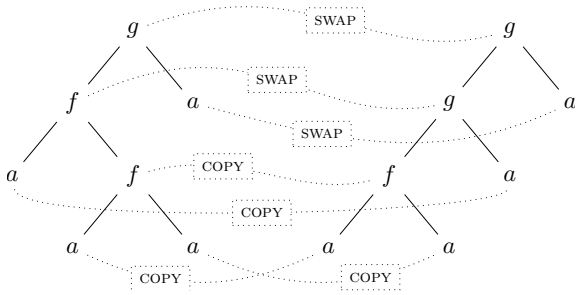

Figure 3: An element of $L(MS)$ in Example 2.

Extending the previous example, one can easily see that all top-down and bottom-up tree transductions can be turned into Millstream systems in a way similar to the construction above. A similar remark holds for many other types of tree transductions known from the literature. Most notably, monadic second-order definable tree transductions (Engelfriet and Maneth, 1999; Engelfriet and Hoogeboom, 2001; Engelfriet and Maneth, 2003) can be expressed as Millstream systems. Since the mentioned types of tree transductions are well studied, and much is known about their algorithmic properties, future research on Millstream systems should investigate the relationship between different types of tree transductions and Millstream systems in detail. In particular, it should be tried to formulate requirements on the interface conditions that can be used to obtain characterisations of various classes of tree transductions. We note here that results of this type would not only be interesting from a purely mathematical point of view, since tree transducers have turned out to be a valuable tool in, for example, machine translation (Knight and Graehl, 2005; May and Knight, 2006; Graehl et al., 2008).

7 Preliminary Results and Future Work

Millstream systems, as introduced in this article, are formal devices that allow to model situations in which several tree-generating modules are interconnected by logical interfaces. In a forthcoming paper (Bensch et al., 2010), we investigate the theoretical properties of regular MSO Millstream systems, i.e., Millstream systems in which the modules are regular tree grammars and the logic used is monadic second-order logic. In particular, we study the so-called *completion problem*. Given a Millstream system with k modules and $l \leq k$ *known trees* $t_{i_1}, \ldots, t_{i_l}$ ($1 \leq i_1 < \cdots < i_l \leq k$), the task is to find a *completion*, i.e., a configuration whose i_jth tree is t_{i_j} for all $j \in [l]$. Thus, if viewed as a pure decision problem, the completion problem corresponds to the membership problem for $L^{M_{i_1} \times \cdots \times M_{i_l}}(MS)$. To be useful in applications, algorithms solving the completion problem should, of course, be required to explicitly construct a completion rather than just answering *yes*.

Let us briefly summarize the results of (Bensch et al., 2010).

1. In general, the completion problem is undecidable for $k - l \geq 2$ even in the case where only the use of first-order logic is permitted. This can be shown by reducing Post's correspondence problem (PCP) to the emptiness problem for a regular FO Millstream system with $k = 2$. The Millstream system constructed is somewhat similar to the one in Example 1, as it establishes bijective correspondences between the nodes of two trees (that

represent the two parts of a solution to a PCP instance).

2. If there are no direct links between unknown trees (i.e., $|\{j_1, \ldots, j_m\} \setminus \{i_1, \ldots, i_l\}| \leq 1$ for each interface symbol $I\colon j_1 \times \cdots \times j_m$), then the completion problem is solvable for all regular MSO Millstream systems.

3. Applying some well-known results, the completion problem is solvable for all regular MSO Millstream systems for which $L(MS)$ is of bounded tree width. Thus, it is of interest to establish conditions that guarantee the configurations in $L(MS)$ to be of bounded tree width. Two such conditions, are given in (Bensch et al., 2010). Roughly speaking, they require that the links respect the structure of the trees. Let us informally describe one of them, called nestedness. Say that a link $I'(u_1, \ldots, u_m)$ is *directly below* a link $I(v_1, \ldots, v_l)$ if there are i, j such that u_j is a descendant of v_i and none of the nodes in between carries a link. Now, fix a constant h. A configuration is *nested* if the roots are linked with each other and the following hold for every link $\lambda = I(v_1, \ldots, v_l)$:

   (a) There are at most h links $I'(u_1, \ldots, u_m)$ directly below λ.

   (b) Each of the nodes u_j in (a) is a descendant of one of the nodes v_i.

   As mentioned above, $L(MS)$ is of bounded tree width if its configurations are nested (with respect to the same constant h).

Nestedness, and also the second sufficient condition for bounded tree width studied in (Bensch et al., 2010) restrict the configurations themselves. While such conditions may be appropriate in many practical cases (where one *knows* what the configurations look like), future research should also attempt to find out whether it is possible to put some easily testable requirements on the interface conditions in order to force the configurations to be of bounded tree width. Note that, since the property of being of tree width at most d is expressible in monadic second-order logic, one can always artificially force the configurations of a given MSO Millstream system to be of bounded tree width, but this is not very useful as it would simply exclude those configurations whose tree width is greater than the desired constant d, thus changing the semantics of the given Millstream system in a usually undesired manner.

Future work should also investigate properties that make it possible to obtain or complete configurations in a generative way. For example, for regular MSO Millstream systems with interface conditions of a suitable type, it should be possible to generate the configurations in $L(MS)$ by generating the k trees in a parallel top-down manner, at the same time establishing the interface links. Results of this kind could also be used for solving the completion problem in an efficient manner. In general, it is clear that efficiency must be an important aspect of future theoretical investigations into Millstream systems.

In addition to theoretical results, a good implementation of Millstream systems is needed in order to make it possible to implement nontrivial examples. While this work should, to the extent possible, be application independent, it will also be necessary to seriously attempt to formalise and implement linguistic theories as Millstream systems. This includes exploring various such theories with respect to their appropriateness.

To gain further insight into the usefulness and limitations of Millstream systems for Computational Linguistics, future work should elaborate if and how it is possible to translate formalisms such as HPSG, LFG, CCG, FDG and XDG into Millstream systems.

Acknowledgments

We thank Dot and Danie van der Walt for providing us with a calm and relaxed atmosphere at Millstream Guest House in Stellenbosch (South Africa), where the first ideas around Millstream systems were born in April 2009. Scientifically, we would like to thank Henrik Björklund, Stephen J. Hegner, and Brink van der Merwe for discussions and constructive input. Furthermore, we would like to thank one of the referees for valuable comments.

References

Suna Bensch and Frank Drewes. 2009. Millstream systems. Report UMINF 09.21, Umeå University. Available at `http://www8.cs.umu.se/research/uminf/index.cgi?year=2009&number=21`.

Suna Bensch, Henrik Björklund, and Frank Drewes. 2010. Algorithmic properties of Millstream systems. In Sheng Yu, editor, *Proc. 14th Intl. Conf. on Developments in Language Theory (DLT 2010)*, Lecture Notes in Computer Science. To appear.

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Christof Löding, Denis Lugiez, Sophie Tison, and Marc Tommasi. 2007. *Tree Automata Techniques and Applications*. Internet publication available at `http://tata.gforge.inria.fr`. Release October 2007.

Mary Dalrymple. 2001. *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics*. Academic Press.

Ralph Debusmann and Gert Smolka. 2006. Multidimensional dependency grammar as multigraph description. In *Proceedings of FLAIRS Conference*, pages 740–745.

Ralph Debusmann. 2006. *Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description*. Ph.D. thesis, Universität des Saarlandes. Available at `http://www.ps.uni-sb.de/~rade/papers/diss.pdf`.

Joost Engelfriet and Henrik Jan Hoogeboom. 2001. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2:216–254.

Joost Engelfriet and Sebastian Maneth. 1999. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154:34–91.

Joost Engelfriet and Sebastian Maneth. 2003. Macro tree translations of linear size increase are MSO definable. *SIAM Journal on Computing*, 32:950–1006.

Zoltán Fülöp and Heiko Vogler. 1998. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer.

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Vol. 3: *Beyond Words*, chapter 1, pages 1–68. Springer.

Jonathan Graehl, Kevin Knight, and Jonathan May. 2008. Training tree transducers. *Computational Linguistics*, 34(3):391–427.

Ray Jackendoff. 2002. *Foundations of Language: Brain, Meaning, Grammar, Evolution*. Oxford University Press, Oxford.

Kevin Knight and Jonathan Graehl. 2005. An overview of probabilistic tree transducers for natural language processing. In Alexander F. Gelbukh, editor, *Proc. 6th Intl. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing 2005)*, volume 3406 of Lecture Notes in Computer Science, pages 1–24. Springer.

Jonathan May and Kevin Knight. 2006. Tiburon: A weighted tree automata toolkit. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *Proc. 11th Intl. Conf. on Implementation and Application of Automata (CIAA 2006)*, volume 4094 of Lecture Notes in Computer Science, pages 102–113. Springer.

Carl Pollard and Ivan Sag. 1994. *Head-Driven Phrase Structure Grammar*. Chicago University Press.

Jerrold Sadock. 1991. *Autolexical Syntax - A Theory of Parallel Grammatical Representations*. The University of Chicago Press, Chicago & London.

Petr Sgall, Eva Hajičová, and Jarmila Panevová. 1986. *The meaning of the sentence in its semantic and pragmatic aspects*. Reidel, Dordrecht.

Mark Steedman. 2000. *The Syntactic Process (Language, Speech, and Communication)*. MIT Press.