Implementing semantic frames as typed feature structures with XMG

Timm Lichte[1] *and Simon Petitjean*[2]
[1] University of Düsseldorf, Germany
[2] Université d'Orléans, LIFO, France

ABSTRACT

This work[1] presents results on the integration of frame-based representations into the framework of eXtensible MetaGrammar (XMG). Originally XMG allowed for the description of tree-based syntactic structures and underspecified representations of predicate-logical formulae, but the representation of frames as a sort of typed feature structure, particularly type unification, was not supported. Therefore, we introduce an extension that is capable of handling frame representations directly by means of a novel `<frame>`-dimension. The aim is not only to make possible a straightforward specification of frame descriptions, but also to offer various ways to specify constraints on types, be it as a connected type hierarchy or a loose set of feature structure con-

Keywords: metagrammar, typed feature structures, models of type constraints, type hierarchy, unification

[1] This article is a substantially revised and extended version of Lichte *et al.* (2013). It has greatly benefited from discussions with Laura Kallmeyer, Rainer Osswald, Christof Rumpf and Yulia Zinova. We also thank both the reviewers of the ESSLLI 2013 workshop on High-level Methodologies in Grammar Engineering (HMGE) and the the reviewers of the JLM for detailed and helpful comments. The work presented in this paper was financed by the Deutsche Forschungsgemeinschaft (DFG) within the CRC 991. It was also financially supported by the scientific network PARSEME (COST Action IC1207) with a travel grant for Simon Petitjean. When finishing this article we learned with great sadness of the passing of our friend and colleague Christof Rumpf. We dedicate this article to his memory.

straints. The presented extensions to XMG are fully operational in a new prototype.

1 INTRODUCTION

Recent work (Kallmeyer and Osswald 2012a,b, 2013; Zinova and Kallmeyer 2012) has shown increasing interest in coupling a frame-based semantics with a tree-based syntax such as Tree Adjoining Grammar (TAG, Joshi and Schabes 1997). While having led to promising results on the theoretic side, it is still unclear how to implement these ideas with existing grammar engineering tools, let alone how to bring them alive in natural language parsing. In this article, we present results on the integration of frame-based representations into the framework of eXtensible MetaGrammar (XMG, Crabbé *et al.* 2013).[2] XMG originally allowed for the description of tree-based syntactic structures and underspecified representations of predicate-logical formulae, but the representation of frames as a sort of typed feature structure, particularly type unification, was not supported. Therefore we extend XMG by a novel `<frame>`-dimension, among other tools, that makes it capable of handling frame representations as formalized in Petersen (2007) and Kallmeyer and Osswald (2013), i.e. as extended typed feature structures, directly.[3] This capability also paves the way for implementing recent work on morphological decomposition, such as in Zinova and Kallmeyer (2012), where morphemes are linked to a frame-semantic representation.

These efforts might seem redundant, given that there exists a multitude of works on dealing with typed feature structures in grammar implementation and parsing, notably in the framework of HPSG (e.g. Carpenter 1992; Götz *et al.* 1997; Malouf *et al.* 2000; Flickinger 2000; Copestake 2002; Carpenter *et al.* 2003). As far as we can see, however, our approach differs from previous work in several respects: first and

[2] `https://sourcesup.renater.fr/xmg/`

[3] The reviewers suggested to focus more on typed feature structures and less on semantic frames. While this is a reasonable advice, we consider the application to be the driving force of this whole endeavour, which also crucially motivates our choice of a typed feature structure logic that is, from our perspective, uncommon in linguistic applications.

foremost, it allows for a choice between minimal and maximal models of type constraints. As a consequence of this, our approach also allows for anonymous types, hence types that result from the conjunction of defined types, but that are not defined themselves in the type signature. We will precisely explicate this distinction in Section 4. Finally, our work aims at providing the grammar writer with multiple means of expression, for example, in permitting him/her to make use of loose type constraints or connected type hierarchies, or both. This degree of flexibility is often not found in other grammar implementation frameworks that support typed feature structures. We hypothesize that other frameworks are mainly concerned with the syntactic component of the grammar, while we focus on the conceptual semantics whose structure (and how it comes about) seems to be much less predetermined.

The paper proceeds as follows. The next section briefly illustrates the grammatical objects that we are concerned with, and Section 3 then shows the proposed factorization, which crucially guides the implementation with XMG and also justifies the employment of frame unification. After this, we explain the formalization of frames as a sort of typed feature structure in Section 4, and the basic concepts of XMG in Section 5. This will be essential to understand the usage and compilation details of the new `<frame>`-dimension, the presentation of which follows in Section 6. Finally, Section 7 concludes the article. Moreover, a complete code example based on analyses from Section 2 and Section 3 is presented in the Appendix.

2 A FRAME-BASED SEMANTICS FOR LTAG

We will use the state-of-the-art work of Kallmeyer and Osswald (2013) on integrating frame semantics into Lexicalized Tree-Adjoining Grammars (LTAG) as a starting point. One important motivation for developing a frame-based semantics for LTAG is found in the straightforward account for the well-established distinction between lexical and constructional contributions to the overall meaning. An example of this sort of contrast is displayed in (1):

(1) a. The ball rolled into the goal.
  b. John rolled the ball into the goal.

Both sentences involve the same verb of directed motion, *rolled*, but the two instances nevertheless differ with respect to the linking of the subject with the conveyed event semantics. While in (1a) the subject *the ball* is the moved object, in (1b) the subject *John* is causing the motion rather than undergoing it. The trigger for this semantic shift seems to be the lack or existence of the direct object (both under the presence of a directional PP), hence the construction type, as this constitutes the crucial syntactic difference between (1a) and (1b).

In the framework of Kallmeyer and Osswald (2013), the syntax of these constructions lies in the scope of the LTAG component. An LTAG consists of a finite set of phrase structure trees, the *elementary trees*, that can be combined (by means of two basic operations, substitution and adjunction) to generate larger trees.[4] Since we are dealing with a lexicalized TAG, each elementary tree must include at least one non-terminal leaf, the *lexical anchor*. Furthermore elementary trees are constrained through the valency properties of their anchors. Usually each non-terminal leaf corresponds to exactly one syntactic argument, and vice versa.

The proposed LTAG analyses of (1a) and (1b), shown in Figure 1, differ with respect to the elementary trees that are associated with the two instances of *rolled*, say the intransitive *rolled*$_{\text{int}}$ and the transitive *rolled*$_{\text{tr}}$:[5] since syntactic arguments are represented as non-terminal leaves, the elementary tree for *rolled*$_{\text{int}}$ lacks the object NP slot that the elementary tree for *rolled*$_{\text{tr}}$ has. The difference in meaning is therefore attributed to the different elementary trees that a given verb may anchor.

Since elementary trees, but not the anchoring verbs, are held responsible for different linking patterns, it is straightforward to abstract away from the concrete anchor by just considering the yet unanchored elementary tree, which is commonly called the *tree template*. An example of such a tree template is shown on the left in Figure 2, wherein the site of lexical insertion, here of *rolled*, is marked by the ⋄-symbol. The

[4] Since in the present paper we mainly focus on single elementary trees, we skip most details of the formalism here. See Joshi and Schabes (1997) or Abeillé and Rambow (2000a) for comprehensive presentations.

[5] Note that dashed arrows indicate combinatorial operations, which in this case only involve substitution, i.e. the rewriting of a leaf node in the target tree.

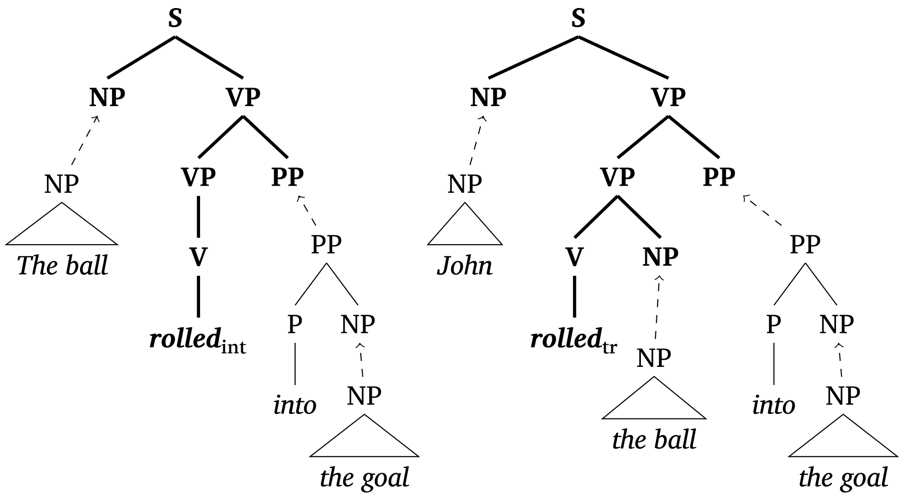
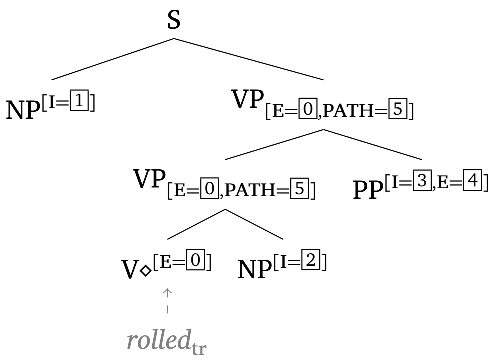


Figure 1: LTAG derivation for (1a) and (1b) with intransitive versus transitive *rolled*



$$\boxed{0}\begin{bmatrix} \textit{causation} & \\ \text{ACTOR} & \boxed{1} \\ \text{THEME} & \boxed{2} \\ \text{GOAL} & \boxed{3} \\ \text{CAUSE} & \begin{bmatrix} \textit{activity} & \\ \text{ACTOR} & \boxed{1} \\ \text{THEME} & \boxed{2} \end{bmatrix} \\ \text{EFFECT} & \boxed{4}\begin{bmatrix} \textit{bounded-translocation} & \\ \text{MOVER} & \boxed{2} \\ \text{GOAL} & \boxed{3} \\ \text{PATH} & \boxed{5} \end{bmatrix} \end{bmatrix}$$

Figure 2: Tree template and frame-semantic representation of the transitive motion construction from Figure 1, taken from Kallmeyer and Osswald (2013, Fig. 26)

complex node labels will be explained presently. The right side of Figure 2 shows the event semantic contribution of the tree template in the format of a typed feature structure, here represented as an attribute value matrix (AVM). Typed feature structures are a common representation format of *frames* (see Petersen 2007), which, according to

Fillmore (1982), Barsalou (1992) and others, are considered a proper representation of mental concepts.[6] As can be seen from the example, features describe semantic participants and components (AGENT, THEME, ...), while feature structures correspond to conceptual objects, restricted by the type (*causation*, *activity*, ...) that they are associated with. The boxed numbers, finally, are *base labels* which serve to mark inequalities and correspondences in the syntax-frame interface. Furthermore, they guide the unification of (subparts of) frames, as can be seen in the next section.

Because tree templates, just as elementary trees, span an extended domain of locality,[7] the linking of positions within the tree template to positions within the frame-semantic representation can be achieved rather directly. In Figure 2 it is indicated by co-occurring boxed numbers. For example, the subject NP-leaf is linked with the ACTOR role(s) of the frame, eventually causing the unification of the ACTOR role and the frame of the substituting NP-tree. Note that the nodes of tree templates carry (non-recursive, non-typed) feature structures, which include, among others, interface features such as I(NDIVIDUAL) and E(VENT).[8] Following the terminology in Kallmeyer and Osswald (2013), we call couples of tree template and frame-semantic representation an *elementary construction*.

The composition of frame representations is moreover guided by a globally defined type hierarchy, which determines (i) the unifiability of types and the resulting type, and (ii) the set of appropriate features

[6] Note that FrameNet (Fillmore 2007), despite being declared as an implementation of Fillmore's frame semantics, deals with flat lexical frame representations that are generally less expressive (Osswald and Van Valin 2014).

[7] The extended domain of locality (EDL) is one of the central properties of the TAG formalism (cf. Joshi *et al.* 1990). It amounts to the capability of arguments to immediately attach to the elementary tree of their governor; see again Figure 1. In connection with TAG, EDL presupposes the availability of the adjunction operation (i.e. the rewriting of inner nodes), in order to account for discontinuity effects such as long distance dependencies.

[8] The approach to let the syntax-semantics interface rely on the unification of interface features can already be found in, e.g., Stone and Doran (1997), Frank and van Genabith (2001), Gardent and Kallmeyer (2003). The proposal for linking nodes of an elementary tree with positions in some semantic representation, and thus to derive syntax and semantics in parallel, dates back at least to Shieber and Schabes (1990).

(and their value types) of a type, the *appropriateness conditions*. Regarding event types, Kallmeyer and Osswald (2013) work with the partial type hierarchy in Figure 3. It has to be read top-down, with

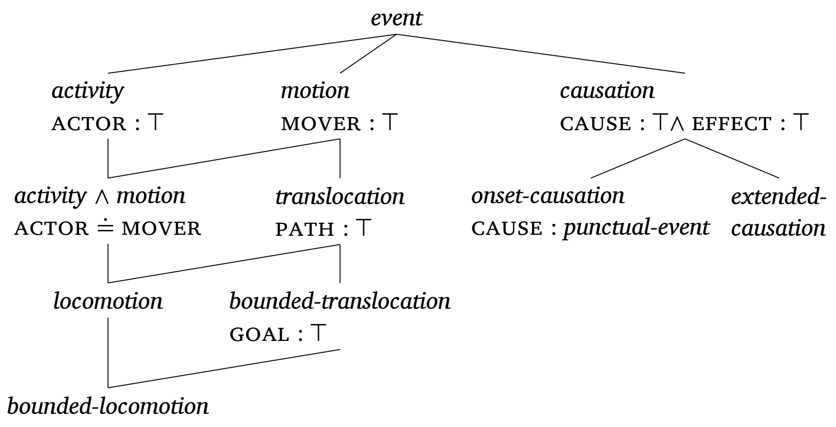


Figure 3: Partial type hierarchy for event types proposed in Kallmeyer and Osswald (2013, Fig. 16).

the more general types dominating the more specific types. Note that the type hierarchy may also contain anonymous types, e.g. *activity* ∧ *motion*, which are usually not available in other frameworks. Roughly speaking, anonymous types make it possible to assign appropriateness conditions to a conjunction of types rather than just to a single type. The details will be explained in Section 4.

In this work we are neither concerned with the unification of frame representations following syntactic composition, nor with the preceding process of lexical insertion that triggers the unification of lexical and constructional frame components (see Kallmeyer and Osswald 2013, Fig. 13), but rather with the metagrammatical framework of XMG. Metagrammars are a tool to describe static elementary constructions such as in Figure 2, consisting of a tree template and a fixed typed feature structure. It might therefore seem unnecessary to employ all aspects of typed feature structures in metagrammars, particularly unification. However, this assumption is not warranted. Unification of types and feature structures is also found in the metagrammatical domain once the factorization of elementary constructions of the kind in Figure 2 is taken into account. This will be shown in the next section.

3 FACTORIZATION OF TREE TEMPLATES AND FRAMES

Richly structured grammatical objects like those in Figure 2 make necessary some kind of metagrammatical factorization, once a large coverage grammar gets compiled and maintained (Xia *et al.* 2010). Metagrammatical factorization is a process to define recurring subcomponents of grammatical objects, which can then be combined in at least two ways: in a transformation-based fashion, known as the metarule approach (Becker 1994, 2000; Prolo 2002), or in a purely constraint-based, monotonic fashion as is the case in XMG (following Candito 1996). In addition to the benefit in terms of grammar engineering, however, Kallmeyer and Osswald (2012a,b, 2013) claim that metagrammar factorization can be also used to reflect constructional analyses in the spirit of Construction Grammar (Kay 2002; Goldberg 2006). By this perspective, both the lexical material and the "constructions" used contribute meaning.

Taking these two aspects into account, Kallmeyer and Osswald (2013) propose to factorize the tree template and the frame in Figure 2 along the lines of Figure 4.[9] Boxes stand for the resulting factors or classes (i.e. classes in the sense of XMG), consisting of descriptions of a tree and a frame fragment. The inclusion relation between boxes is to be understood as a representation of inheritance or instantiation, so that the class of the comprising box inherits from, or instantiates, the class of the included box. Double edges (indicating identity constraints over nodes or base labels), dashed edges (non-strict dominance), $\prec^*$ (non-strict precedence), and $\vee$ (disjunction) are elements of the description language. Figure 4 then illustrates that the tree-frame couple in Figure 2 is a model of the class n0Vn1pp(dir), which combines the classes n0Vn1 and DirPrepObj. Combining two classes essentially means that all associated information is unified, from which a minimal model is resolved (see Section 5). Note that Figure 4 shows only a part of the proposed factorization. For example, the class n0Vn1, also taken from Kallmeyer and Osswald (2013, Fig. 4), results from combining three other classes (Subj, VSpine, DirObj), as shown in Figure 5.[10] Furthermore note that the class n0Vn1pp(dir) bears a con-

[9] The boxes-and-pipes notation in Figures 4 and 5 is of our invention.

[10] Kallmeyer and Osswald (2013) conjecture that class n0V has exactly one

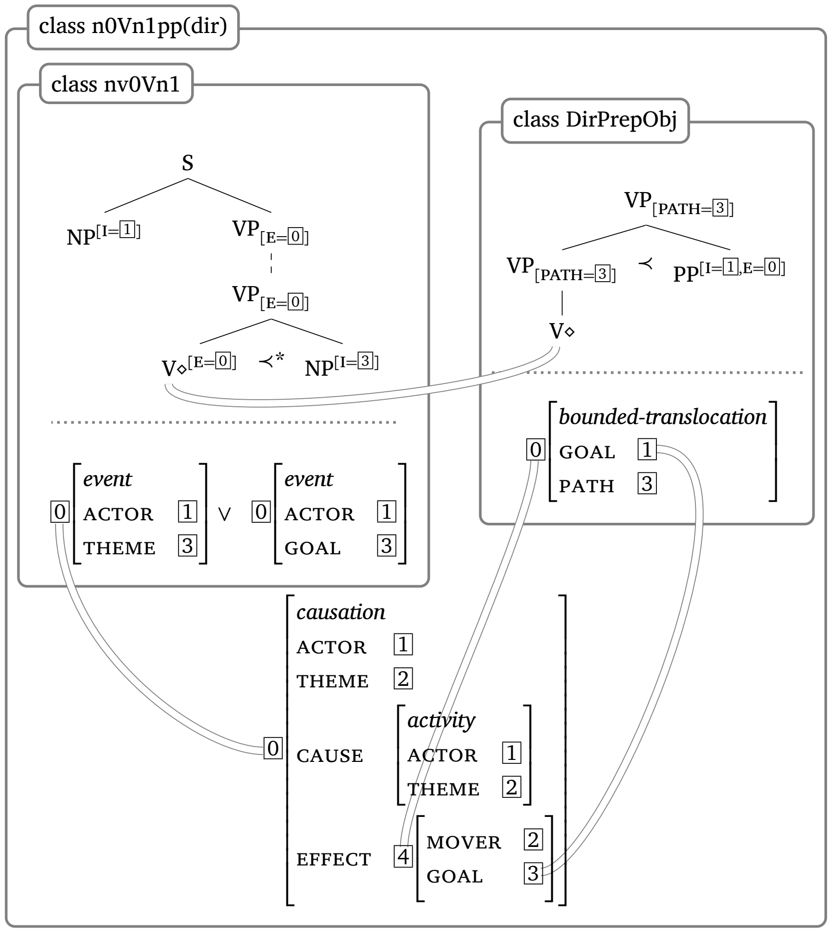


Figure 4: Metagrammatical factorization of the elementary construction from Figure 2. Boxes stand for the resulting factors or classes. Double edges indicate identity constraints, dashed edges indicate non-strict dominance, $\prec^*$ is non-strict precedence, and $\vee$ is disjunction.

minimal model, which only holds, however, if minimal models are said to consist of the smallest set of nodes that satisfy the description. In XMG, by contrast, minimal models just consist of a set of nodes mentioned in the description, and nothing more. Hence, from point of view of XMG, n0V has two minimal models, one with one VP node, and one with two VP nodes. In order to resolve only the former one, one could additionally apply polarization, or “colours” (Crabbé and Duchier 2005), onto the nodes. Fortunately, the question how many minimal models there are is irrelevant for the point made here and throughout the article.

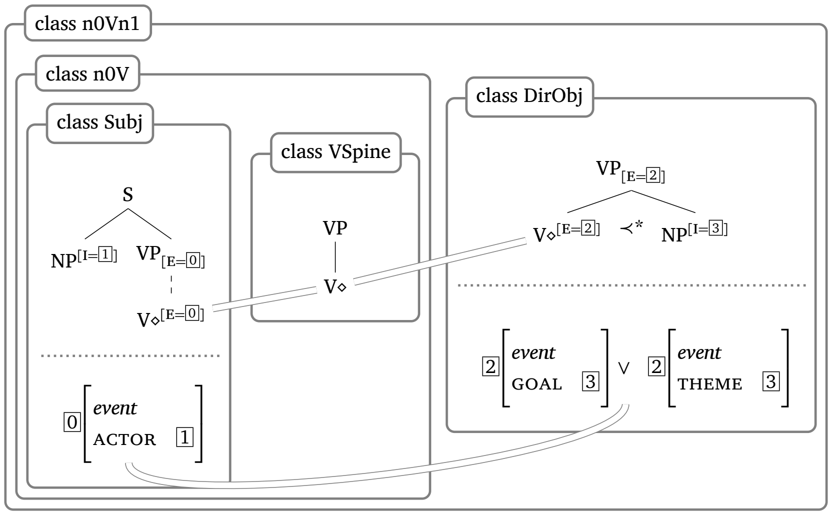


Figure 5: Metagrammatical factorization for the transitive elementary construction.

structional facet: it only contributes to the frame representation, but no tree descriptions.

Now the question is whether the combination of frame representations, following the combination of two classes, should be considered a unification of typed feature structures. It is not hard to see that this is justified. A particularly good example is found in the combination of the frame representations of n0Vn1 and n0Vn1pp(dir) in Figure 4. Even though they have different types, namely *event* and *causation*, the resulting type is supposed to be *causation*. According to the type hierarchy in Figure 3, *causation* is a subtype of *event*. Hence this is in line with regular type unification, but exceeds a plain union of feature structures. Therefore the unification of typed feature structures should be already supported at the metagrammatical level. The same holds for the verification of appropriateness conditions, of course. Note that, as can be seen from Figure 4, the flexibility of frame unification is fostered by base labels when unifying the frame of DirPrepObj with a subpart of the frame of n0Vn1pp(dir).

The graphic representation of the metagrammatical factorization in Figure 4 and Figure 5 remains at a rather informal level and the question arises: how could this be translated into XMG code? Since the original XMG did not give a satisfying answer due to reasons of usability and completeness (which we will cover in Section 5) we will develop and justify a new `<frame>`-dimension in Section 6. But first we will review the underlying notions: frames as typed feature structure with base labels, type hierarchies and unification.

4 BASE-LABELLED TYPED FEATURE STRUCTURES: FORMAL DEFINITIONS

In what follows we largely, but not exclusively, adhere to the definitions in Kallmeyer and Osswald (2013), while streamlining them according to our terminology and taste. What is not needed from Kallmeyer and Osswald (2013), however, are (non-functional) relations, because they do not appear in the frame representations of elementary constructions, other than, e.g., in the frame representations of the anchor lexicon (Kallmeyer and Osswald 2013, Fig. 13).[11] So we will ignore the notion of (non-functional) relations, and instead dwell on type inference and the minimal and maximal models of feature structure constraints. It is this choice between minimal and maximal models, as well as the general availability of anonymous types, which makes the presented extension to XMG particularly flexible at describing a type system. And it is this flexibility, among other things, which sets our extension to XMG apart from current frameworks for HPSG.

Let us start with the building blocks of typed feature structures, which are settled in the *signature*:

Definition 1 (Signature) *A signature is a tuple $\langle A, T, B\rangle$ with a finite set of attributes (or features) A, a finite set of elementary types T, and an infinitely countable set of base labels B.*

The set of elementary types is accompanied by special types $\top$ and $\bot$ that are useful in feature structure descriptions and feature structure constraints (see below). $\top$ is the most general type, unifiable with every other type, whereas $\bot$ is unifiable with none of them. Within Boolean expressions, $\top$ corresponds to 'true' and $\bot$ to 'false'. *Base labels* are commonly boxed natural numbers, thus $B = \{\boxed{0}, \boxed{1}, \boxed{2}, \ldots\}$.

Based on a given signature, typed feature structures are defined as follows:

[11] Kallmeyer and Osswald (2013) make use of the part-of relation when dealing with directional prepositions such as *to*, *into*, and *along*.

Definition 2 (Base-labelled typed feature structure) *Given a signature $\langle A, T, B\rangle$, a base-labelled typed feature structure is a tuple $\langle V, \delta, \tau, \beta\rangle$ with*

- V*, a finite set of nodes,*
- $\delta : (V \times A) \rightarrow V$*, a partial transition function,*
- $\tau : V \rightarrow 2^T$*, a total typing function, and*
- $\beta : B \rightarrow V$*, a partial base-labelling function.*

We call types in 2^T, i.e. elements of the powerset of T, *conjunctive types* in order to distinguish them from elementary types.

We follow Kallmeyer and Osswald (2013) in that we do not specify a type hierarchy immediately, but treat it as a model of feature structure constraints, i.e. generalized feature structure descriptions.[12] Therefore the following notations can be directly borrowed from Kallmeyer and Osswald (2013, (3)), omitting only those parts of their definition that deal with relations. Note that we extend δ by feature paths in the usual way.[13]

Definition 3 (Unlabelled feature structure description) *Let $\langle V, \delta, \tau, \beta\rangle$ be a feature structure over the signature $\langle A, T, B\rangle$ with $v, w \in V$, $t \in 2^T$ and $p, q \in A^+$, then the satisfaction relation $\models$ between nodes and feature structure descriptions is defined as follows:*

- $v \models t$ *iff* $t \in \tau(v)$
- $v \models p : t$ *iff* $\delta(v,p) \models t$
- $v \models p \doteq q$ *iff* $\delta(v,p) = \delta(v,q)$
- $\langle v, w\rangle \models p \triangleq q$ *iff* $\delta(v,p) = \delta(w,q)$

As such, unlabelled feature structure descriptions apply to nodes of a feature structure. For example, the root node of the causation frame in Figure 4 satisfies the following conjunction of descriptions:

(2) *causation* ∧ ACTOR $\doteq$ CAUSE ACTOR ∧ THEME $\doteq$ CAUSE THEME ∧ THEME $\doteq$ EFFECT MOVER ∧ CAUSE : *activity* ∧ EFFECT GOAL : $\top$

By adding base labels, we can explicitly assign descriptions to nodes within a feature structure. Therefore, in contrast to unlabelled ones,

[12] However, XMG also allows one to specify a type signature explicitly; see Section 6.

[13] $\delta(v, p\ a)$ with $p \in A^+$ and $a \in A$ is a shorthand for $\delta(\delta(v,p), a)$.

the labelled feature structure descriptions are satisfied by feature structures as a whole:[14]

Definition 4 (Labelled feature structure description) *Let $F = \langle V, \delta, \tau, \beta \rangle$ be a feature structure over the signature $\langle A, T, B \rangle$ with $p, q \in A^+$ and $l, k \in B$, and let ϕ be an unlabelled feature structure description. The satisfaction relation $\models$ between feature structures and labelled feature structure descriptions is defined in the following way:*

- $F \models l\ \phi$ *iff* $\beta(l) \models \phi$
- $F \models l\ p \triangleq k\ q$ *iff* $\langle \beta(l), \beta(k) \rangle \models p \triangleq q$

The unlabelled descriptions in (2) can be straightforwardly labelled in order to be satisfied by the causation frame as a whole:

(3) $\boxed{0}$ *causation* $\wedge$ $\boxed{0}$ ACTOR $\doteq$ CAUSE ACTOR $\wedge$ $\boxed{0}$ THEME $\doteq$ CAUSE THEME $\wedge$ $\boxed{0}$ THEME $\doteq$ EFFECT MOVER $\wedge$ $\boxed{0}$ CAUSE : *activity* $\wedge$ $\boxed{0}$ EFFECT GOAL : $\top$

Note that $\boxed{0}$ ACTOR $\doteq$ CAUSE ACTOR is equivalent with $\boxed{0}$ ACTOR $\triangleq$ $\boxed{0}$ CAUSE ACTOR (see Kallmeyer and Osswald 2013, fn. 14). We adopt the convention to write l instead of $l\ \epsilon$ for some label l. This allows us to write $\boxed{1} \triangleq \boxed{2}$ for expressing the value identity of labels $\boxed{1}$ and $\boxed{2}$.

Concerning the subsumption relation on feature structures, we can transfer the fairly standard definition from Kallmeyer and Osswald (2013), which they extend by base labels.

Definition 5 (Subsumption, $\sqsubseteq$) *Given feature structures $F_1 = \langle V_1, \delta_1, \tau_1, \beta_1 \rangle$ and $F_2 = \langle V_2, \delta_2, \tau_2, \beta_2 \rangle$ over the signature $\langle A, T, B \rangle$, F_1 subsumes F_2, if there is a function $h : V_1 \to V_2$ so that*

- *if $\delta_1(v, a)$ is defined for $v \in V_1$ and $a \in A$, then $\delta_2(h(v), a) = h(\delta_1(v, a))$;*
- *for every $v \in V_1$, $\tau_1(v) \subseteq \tau_2(h(v))$;*
- *if $\beta_1(l)$ is defined for $l \in B$, then $\beta_2(l) = h(\beta_1(l))$.*

As usual, the definition of unification builds on subsumption:

[14] From the general shape of labelled feature structure descriptions it follows that they can only describe feature structures where every node is reachable from some labelled node via a (potentially empty) attribute path. In fact, this is why these labels are called base labels.

Definition 6 (Unification, $\sqcup$) *Let F_1, F_2, F_3 be feature structures. F_3 is the result of the unification of F_1 and F_2, iff F_3 is the least specific feature structure such that $F_1 \sqsubseteq F_3$ and $F_2 \sqsubseteq F_3$.*

The specificity of feature structures is determined by the number of nodes, the specificity of the assigned types and the size of the base labelling function. Note that, in cases of unification such as in Figure 4, feature structures (and corresponding interface variables) are relabelled beforehand, so that they come with disjoint sets of labels. The identity of certain labels can then be imposed through additional identity statements.

Feature structure constraints consist of universally quantified, unlabelled features structure descriptions, i.e. descriptions that hold for every node in the feature structure. Kallmeyer and Osswald (2013) restrict them to Horn clauses for reasons of tractability.[15]

Definition 7 (Feature structure constraint) *Given unlabelled feature structure descriptions $\phi_1, \ldots, \phi_n, \psi$, a feature structure constraint is $\phi_1 \wedge \ldots \wedge \phi_n \preceq \psi$, which is equivalent to the universally quantified implication $\forall(\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi)$ where quantification is over the nodes of the described feature structure.*

The following list shows different kinds of feature structure constraints (mostly taken from Kallmeyer and Osswald 2013, (12)):[16]

(4) a. *activity* $\preceq$ *event*
b. *causation* $\preceq \neg$*activity* $\quad\Leftrightarrow\quad$ *causation* $\wedge$ *activity* $\preceq \perp$
c. *locomotion* $\preceq$ *activity* $\wedge$ *translocation*
$\Leftrightarrow$ (*locomotion* $\preceq$ *activity*) $\wedge$ (*locomotion* $\preceq$ *translocation*)
d. *activity* $\preceq$ ACTOR: $\top$
e. AGENT: $\top \preceq$ AGENT $\doteq$ ACTOR

The first three feature structure constraints in (4a)–(4c) consist of descriptions over types, which is why we call them *type constraints*. The simplest type constraint in (4a) relates two elementary types and cor-

[15] Horn clauses are disjunctive clauses with at most one non-negative literal, all others being negative. Hence they are of the form $\neg\phi_1 \vee \ldots \vee \neg\phi_n \vee \psi$, which has the implicational equivalent $\phi_1 \wedge \ldots \wedge \phi_n \rightarrow \psi$.

[16] Note that $\Leftrightarrow$ in (4) is not part of the description language but indicates the equivalence of descriptions.

responds to an ISA-relation, namely *activity* being also an *event*. An ISNOTA-relation can be expressed by type constraints of the kind in (4b). Note that the use of negation in the consequent is equivalent to a clause with a conjunctive antecedent and a ‘false’ ($\bot$) consequent. Not only can the antecedent be complex, but also the consequent, even though this is reducible to a conjunction of simpler implications, as shown in (4c). The last two examples in (4d) and (4e) contain attribute-value terms. Constraint (4d) represents an *appropriateness condition*, namely a condition on the type *activity* concerning the value type of its attribute ACTOR. Finally, (4e) adds a constraint about the token identity of the values of the attributes AGENT and ACTOR (if AGENT has a value). Constraints of this sort, where every conjunct consists of a feature-value description or a path equation, can be characterized as *feature-value constraints*.[17]

A crucial decision concerns the model of feature structure constraints, as it can be seen as maximal or minimal. The difference is made visible in Figure 6 with type hierarchies based on $\preceq$.

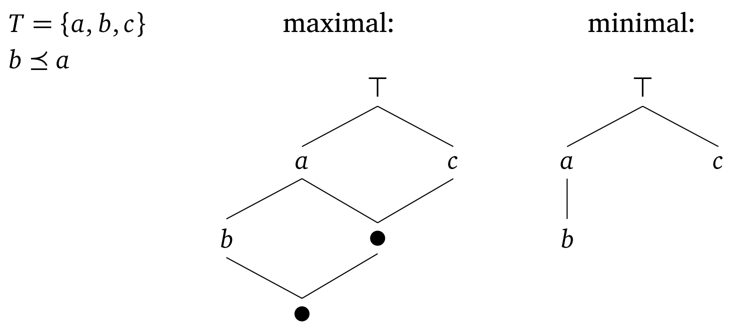


Figure 6: Examples of a maximal and a minimal model of type constraint

Roughly speaking, a maximal model of type constraints allows for *anonymous types* (indicated with dots in Figure 6) as long as they are not explicitly excluded. With minimal models it is the other way

[17] Other sorts of feature structure constraints, for example those in (i), are not discussed in Kallmeyer and Osswald (2013):

(i) a. $p : t_1 \preceq t_2$
b. $p : t_1 \wedge t_2 \preceq t_3$

We therefore concentrate on feature structure constraints in the shape of type constraints, appropriateness conditions and feature-value constraints.

around: anonymous types are forbidden as long as they are not explicitly allowed. In both cases they could be introduced (or excluded) by constraints such as in (5a), where the antecedent consists of a conjunction of elementary types. In Figure 6, $a \wedge c$ and $a \wedge b \wedge c$ are anonymous types. It is shown in (5b) that these anonymous types can bear feature-value constraints as well.[18]

(5) a. $a \wedge c \preceq \top$

b. *action* $\wedge$ *motion* $\preceq$ ACTOR $\doteq$ MOVER
(Kallmeyer and Osswald 2013, (12e))

The preceding remarks were concerned with elementary types. However, as we learned in Definition 2, the typing function τ assigns sets of elementary types, the conjunctive types. They are included for good reason, since they help to treat elementary types and anonymous types in a uniform way, and eventually to facilitate the definition and implementation of unification.[19] In the following, we treat conjunctive types as a model of type constraints, and, on this basis, explicate what it means to be minimal or maximal.

Definition 8 (Model of type constraints) *Given type constraints TC over a signature $\langle A, T, B\rangle$, a set $\hat{T}$ of conjunctive types over T is a model of TC, if $\hat{T}$ satisfies the type constraints from TC in the following way:*

- $\hat{T} \models t_1 \wedge \ldots \wedge t_n \preceq \bot$ *iff* $\{t_1, \ldots, t_n\} \not\subseteq \hat{t}$ *for all* $\hat{t} \in \hat{T}$;
- $\hat{T} \models t_1 \wedge \ldots \wedge t_n \preceq t_m$ *iff if* $\{t_1, \ldots, t_n\} \subseteq \hat{t}$, *then* $\{t_m\} \subseteq \hat{t}$.

The model $\hat{T}$ of given type constraints is said to be *maximal*, if $\hat{T}$ is the largest set of conjunctive types that satisfies them. On the other

[18] Note that anonymous types are generally ruled out in major implementation tools for HPSG such as LKB (Copestake and Flickinger 2000; Copestake 2002) and TRALE (Götz *et al.* 1997; Carpenter *et al.* 2003), from which it follows that models from type constraints are always minimal.

[19] Note however that Carpenter (1992, 23–25) defines conjunctive types differently as he adds the condition that p, but not q, is included whenever the relation $p \preceq q$ is considered. Hence, following his conception, there is no co-occurrence of elementary types and their elementary subtypes within a conjunctive type. On the other hand, our definition of conjunctive type rather coincides with the notion of "conjunctive concepts" in Carpenter and Pollard (1991), or with the notions of "entity types" and "generic entities" in Osswald (2003, 24f).

hand, $\hat{T}$ is said to be the *minimal* model, if $\hat{T}$ includes just those conjunctive types of the maximal model that correspond (i) to elementary types, and (ii) to conjunctive types that make up the left side (i.e. the condition) of a type constraint.

The type hierarchies from Figure 6 for elementary types are repeated in Figure 7 for conjunctive types. They are now based on $\subseteq$ rather than $\preceq$. We say that conjunctive type $\hat{t}_1$ is the subtype of $\hat{t}_2$, if $\hat{t}_1 \supset \hat{t}_2$. The definition of the most general common subtype is equally simple.

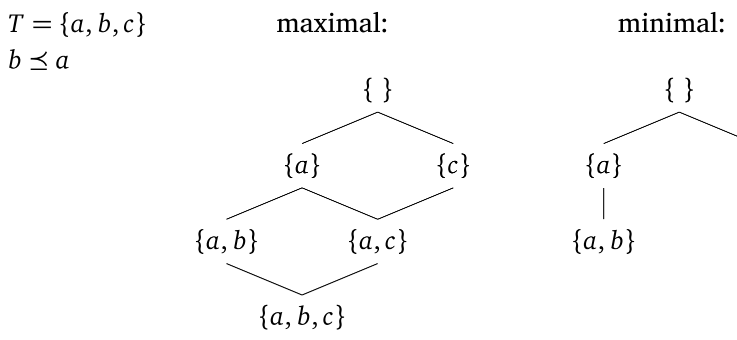


Figure 7: Example of maximal and minimal models of type constraints over conjunctive types

Definition 9 (Most general common subtype) *The most general common subtype of two conjunctive types $\hat{t}_1, \hat{t}_2 \in \hat{T}$ is the smallest set $\hat{t}_3 \in \hat{T}$ such that $\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_3$.*

It is easy to see that, whenever type constraints have the shape of Horn clauses, the maximal model over conjunctive types $\hat{T}$ forms a meet semi-lattice (a bounded complete partially ordered set) $\langle \hat{T}, \subseteq \rangle$. Hence, there is a most general common subtype for any two unifiable types. This does not necessarily hold for minimal models that are confined to elementary and certain anonymous types.[20] In order to obtain uniqueness of the most general common subtype for any two unifiable types, it can be necessary to add further anonymous types to $\hat{T}$ (the Dedekind-McNeille completions). Theses completions are computed by the compiler that will be described in Section 6.3.2.

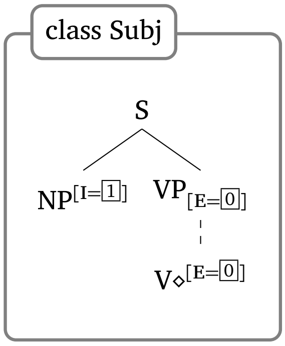
[20] For example, say $T = \{a, b, c, d\}$, $TC = \{c \preceq a \wedge b,\ d \preceq a \wedge b\}$, then $\hat{T} = \{\{\}, \{a\}, \{b\}, \{a, b, c\}, \{a, b, d\}\}$ satisfies TC and only contains the conjunctive types that correspond to elementary types in T. However, $\{a\}$ and $\{b\}$ have more than one most general common subtype in $\hat{T}$, namely $\{a, b, c\}$ and $\{a, b, d\}$.

A conjunctive type $\hat{t}$ corresponds to an elementary type t, if $\hat{t}$ is the smallest set that contains t according to the model. Otherwise, if there is no t to which $\hat{t}$ corresponds, $\hat{t}$ is an anonymous type. Similarly, $\hat{t}$ corresponds to some conjunctive type $\hat{t}'$, if $\hat{t}$ is the smallest set of a model that contains $\hat{t}'$. Moreover, it can be useful to also express subtype relations among elementary types $t_1, t_2 \in T$. We say that t_1 is the subtype of t_2, if for $\hat{t}_1, \hat{t}_2 \in \hat{T}$ that correspond to t_1 and t_2 it holds that $\hat{t}_1$ is a subtype of $\hat{t}_2$.

Before proceeding to the next section, two further aspects of the feature logic used here and in Kallmeyer and Osswald (2013) should be mentioned. Firstly, it cannot account for reverse type constraints, which refer to dominating nodes (Rainer Osswald, personal communication, July 23, 2014). A case in point is the set of relational concepts such as the *mother* example from Petersen and Osswald (2014, Fig. 11.5), where *mother* is said to constrain the existence of a dominating node, connected with a MOTHER edge. Hence, if *mother* was treated as a type (a subtype of, e.g., *person*), this constraint could not be expressed, at least not as a part of its appropriateness conditions. Secondly, what seems to be missing so far from the definition in Petersen (2007) is the notion of the central node of a frame, i.e. the node that tells us what the frame "represents" or "refers to". We think, however, that the notion of central nodes is reflected, and generalized, in the present formalization by the notion of base labels. By reappearing in the interface features on the syntactic side, they serve to connect frame nodes with linguistic entities, and nothing else seems to be expressed by central nodes.

5 A BRIEF INTRODUCTION TO XMG

XMG (eXtensible MetaGrammar, Crabbé *et al.* 2013) stands both for metagrammatical descriptions and the compiler for these descriptions. Such descriptions are organized into classes that can be reused (i.e. "imported" or instantiated) by other classes. Borrowing from object oriented programming, classes are encapsulated, which means that each class can handle the scopes of their variables explicitly, by declaring variables and choosing which ones to make accessible for (i.e. to "export to") other instantiating classes. The namespace of a class is



```
class Subj
...
<syn>{
  node ?S [cat=s];
  node ?SUBJ [cat=np,
              top=[i=?1]];
  node ?VP [cat=vp,bot=[e=?0]];
  node ?V (mark=anchor)
          [cat=v,top=[e=?0]];
  ?S->?SUBJ; ?S->?VP; ?VP->*?V;
  ?SUBJ>>?VP
}
...
```

Figure 8: The <syn>-dimension of class Subj

then composed of the declared variables and all the variables exported by the imported classes.

Dimensions are the crucial elements of a class. They can be equipped with specific description languages and are compiled independently, thereby enabling the grammar writer to treat the levels of linguistic information separately. The standard dimensions are <syn> for the syntax, and <sem> for the semantics.[21]

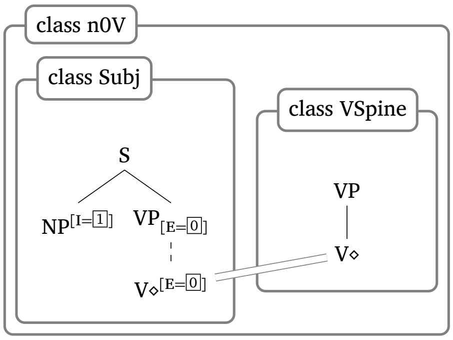
The <syn>-dimension allows one to describe TAG tree templates (or fragments thereof). An example is shown in Figure 8 for the <syn>-dimension of class Subj from Figure 4. It includes two sorts of statements, namely those like '`node ?S [cat=s]`' that instantiate nodes of the trees, and those like '`?S->?SUBJ`' which determine the relative position of two nodes in the trees by referring to dominance and linear precedence.[22] Note that variable names are prefixed with a question mark ('?'). The <sem>-dimension, on the other hand, includes descriptions of a different language, for which a different compiler is used. Since this could be a candidate for hosting frame descriptions, we will have a look at <sem> more closely below. Different as they may be, one crucial commonality of all the dimensions pertains to the joint access

[21] Crabbé *et al.* (2013, 601) also mention the fairly technical dimension <dyn>, more commonly called *interface*, which helps to express the coreference of variables from different dimensions. Recently Duchier *et al.* (2012) have introduced a dimension for morphology; see also Lichte *et al.* (2013).

[22] There is also available a notational alternative with bracket structure.

to local variables declared in the same class. These shared variables constitute a direct interface between otherwise separated dimensions.

The combination of classes takes place outside the `<syn>`- and `<sem>`-dimensions. Figure 9 shows an example where the two classes Subj and VSpine are reused by the class n0V. First Subj and VSpine are instantiated and assigned a variable, then the encapsulated, yet exported, variables from Subj and VSpine can be accessed via the dot operator (e.g. to impose identity).



```
class n0V
...
?Subj = Subj[];
?VSpine = VSpine[];
?Subj.?V = ?VSpine.?V;
...
value n0v
```

Figure 9: Example of the combination of classes

When the metagrammar is compiled, first a set of descriptions for each class under evaluation (triggered by `value` statements such as in Figure 9) is accumulated, and then the accumulated descriptions are resolved to yield minimal models. In the case of `<syn>`, the solver computes tree templates as minimal models, which is to say that only those nodes that are mentioned in the description are included. The final result can be explored with a viewer, or exported as an XML file in order to use it for parsing (e.g. with the TuLiPA parser, Kallmeyer *et al.* 2008).

Frame descriptions in the `<sem>`*-dimension?*

As mentioned before, the `<sem>`-dimension is designed to contain underspecified, flat formulae of predicate logic (borrowing from Bos 1996). In fact, it is rather straightforward to reformulate frame descriptions in first-order predicate logic (Kallmeyer and Osswald 2013, Sec. 3.3.3). Concerning the signature, attributes can be represented with two-place predicates, while types can be seen as one-place predicates. The functionality of attributes is imposed by the following axiom for all $a \in A$, given some signature $\langle A, T \rangle$:

(6) $\forall x \forall y \forall z (a(x,y) \wedge a(x,z) \rightarrow y = z)$
(Kallmeyer and Osswald 2013, (6a))

The reformulation of feature structure descriptions and feature structure constraints is equally unproblematic:

(7)

a.	$p : t$	$\lambda x \exists y (p(x,y) \wedge t(y))$
b.	$p \doteq q$	$\lambda x \exists y (p(x,y) \wedge q(x,y))$
c.	$a\ p$	$\lambda x \lambda z \exists y (a(x,y) \wedge q(y,z))$

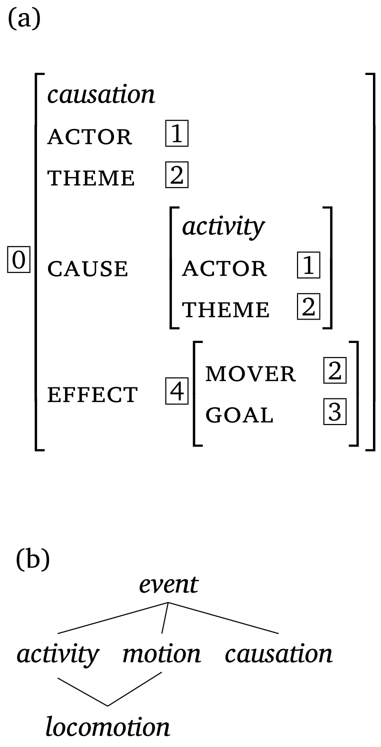
(Kallmeyer and Osswald 2013, (7))

(8)

a.	$t_1 \preceq t_2$	$\forall x (t_1(x) \rightarrow t_2(x))$
b.	$t_1 \preceq p : t_2$	$\forall x \exists y (t_1(x) \rightarrow p(x,y) \wedge t_2(y))$
c.	$t \preceq p \doteq q$	$\forall x \exists y (t(x) \rightarrow p(x,y) \wedge q(x,y))$

Following this approach, a frame representation such as $\boxed{0}$ [ACTOR $\boxed{1}$] would be translated into the two-place predicate *actor*$(?0, ?1)$, using regular XMG variables. A more detailed example based on the class n0Vn1pp(dir) is shown in Figure 10.

While the reformulation of frame descriptions via two-place predicates is straightforward, it is far less obvious how to account for feature structure constraints and the axiom of functionality. As far as type constraints and the type hierarchy are concerned, there seems to be a chance to simulate them with sets of type predicates. This approach is pursued in Figure 10. The construction of those *type simulating sets* (TSSs), as we call them, could proceed as follows: given a signature $\langle A, T \rangle$ and a type hierarchy $\mathcal{T}$ such as the one in Figure 10, we say that $t \in T$ is simulated by the minimal set of predicates $P_t(?X)$ for some variable $?X$, if $P_t(?X)$ is assembled in the following way: for every $t' \in T$, if t' reflexively and transitively dominates t in $\mathcal{T}$, then $t'(?X) \in P_t(?X)$; else if t and t' have no common subtype, then $\sim t'(?X) \in P_t(?X)$, where '$\sim$' stands for negation. To give an example, $P_{locomotion}(?X)$ for the type *locomotion* in the type hierarchy of Figure 10 would be the set $\{activity(?X), motion(?X), \sim causation(?X), locomotion(?X)\}$. It is easily seen that the size of some $P_t(?X)$ crucially depends on the position of t in $\mathcal{T}$, and on the size of T. Note that TSSs do not fully match conjunctive types, due to the insertion of negated type predicates.



(c)

```
class n0Vn1pp(dir)
...
<sem>{
  actor(?0,?1);
  theme(?0,?2);
  cause(?0,?5);
  actor(?5,?1);
  theme(?5,?2);
  effect(?0,?4);
  goal(?4,?3);

  %% causation type
    event(?0);
   ~activity(?0);
   ~motion(?0);
   ~locomotion(?0);
    causation(?0);
  %% activity type
    event(?5);
    activity(?5);
   ~causation(?5)
}
...
```

Figure 10: The feature structure of n0Vn1pp(dir) (repeated from Figure 4), the global type hierarchy (partially repeated from Figure 2), and its reformulation inside the <sem>-dimension

One basic problem of this approach is that so far XMG does not interpret the predicates of the <sem>-dimension, but merely accumulates them for later use. Hence XMG allows for, e.g., the coexistence of predicates $theme(x_1, x_2)$ and $theme(x_1, x_3)$ with $x_2 \neq x_3$, which conflicts with the required functionality of feature predicates. But even if XMG was enhanced to verify the functionality of predicates, at least three disadvantages would remain: (i) TSSs have to be provided by the grammar writer, (ii) they have to be included in the XMG descriptions as a whole, and (iii) unifying sister types with a common subtype will yield a TSS that does not immediately reveal the elementary type of the common subtype. The latter disadvantage might be more of an aesthetic kind, but the first and the second one clearly have an impact on usability. Modifying the type hierarchy in the context of a large grammar would make necessary a meta-metagrammar, that would automatically recompute the TSSs and adapt the parts of the XMG descriptions, where TSSs were used. Rather than considerably modifying and extending the solver of the `<sem>`-dimension, let alone the available description language, we present a novel `<frame>`-dimension in the next section, which is closely adjusted to the peculiarities of frame representations and frame composition.

6 THE IMPLEMENTATION OF TYPED FEATURE STRUCTURE DESCRIPTIONS AND CONSTRAINTS

Implementation entails two operations: specification and compilation. We will deal with the first one in Section 6.2, when introducing specification languages for feature structure descriptions and constraints. The compilation of frame models based on these specifications is then covered in Section 6.3. First, however, the technical prerequisites for extending the XMG compiler are to be outlined.

6.1 *Architecture and extensibility of XMG*

The XMG project started in 2003 with the goal of providing a means to write large scale tree-based grammars, i.e. Tree Adjoining Grammars and Interaction Grammars (Perrier 2000). Originally, the compiler was written in Oz/Mozart, a language which is not maintained anymore. For this reason, and in order to build a compiler more in line with the project's ambitions (regarding modularity and extensibility), it was necessary to restart the implementation from scratch. The new implementation started in 2010 and is sometimes called XMG-NG or XMG2.[23] The compiler is now written in YAP (Yet Another Prolog) with bindings to Gecode for solving constraints. The extensibility is provided by automatic code generation using Python.

With this new version, an XMG compiler can be built from a combination of elementary compiler units. These elementary units are

[23] See `https://sourcesup.renater.fr/xmg/`.

called bricks, and correspond to the set of compiling steps of a metagrammatical language. A brick can correspond to a description language (e.g. the TAG description language of the <syn>-dimension), to a subpart thereof (e.g. the feature structure language used by the <syn>-dimension), or to a solver (e.g. the tree solver used by the same dimension). Every part of the compiler is a brick, even the control language allowing one to express conjunction and disjunction, and a compiler is built by picking bricks and plugging them together. In this process, the parser for a new metagrammatical language, and all the other compiling steps, can be automatically assembled according to the way the bricks are plugged together, to generate a whole compiler for this language. The idea is that every brick holds a fragment of compiler, dedicated to a fragment of language (described by a set of context-free rules). Whilst combining these language fragments to build a full language, the compiler fragments are also contributed to a full compiler.

Hence, when extending XMG by components that are able to handle typed feature structures, we can take advantage of the compiler's modularity to add another module, dedicated to this new task. Such a module, for example a new dimension, can be equipped with a dedicated specification language and compiler.

6.2 *Specification languages*

The design of the specification languages that we present in the following subsections is guided by certain goals, ideas and examples. Firstly, we try to adhere to the notation in the definitions in Section 4 as closely as possible. Secondly, we try to remain consistent with the coding style that already exists in XMG, though being largely unrestricted in principle from a technical point of view. Thirdly, and most importantly, we aim at specification languages that are inherently consistent, lightweight, transparent, and at the same time flexible. In doing so, we share certain elements from specification languages proposed in other work, but, as far as we know, our combination of these elements has not been presented elsewhere.

A complete code example based on the type hierarchy from Section 2 and the factorization from Section 3 is presented in the Appendix.

6.2.1 Specification of the signature

For the specification of the signature, that is to say attributes, types and base labels, the global fields `frame_types` and `frame_attributes` are available:

```
frame_types = {event,activity,motion,causation,...}
frame_attributes = {actor,theme,goal,...}
```

Base labels correspond to XMG variables and do not need to be declared globally.

6.2.2 Specification language for feature structure descriptions

Feature structure descriptions are specified within the `<frame>`-dimension of a class, which comes with a dedicated specification language and compiler. The mode of operation of the compiler is detailed below in Section 6.3.1. We make use of the following description language, which is basically a simple bracket notation:[24]

```
<frame>{Descriptions;Descriptions;...}
Descriptions ::= var? '[' Description (',' Description)* ']'
Description  ::= type | PathEquation | AVPair
PathEquation ::= attr+ '=' var? attr+ (':' Value)?
AVPair ::= attr+ ':' Value
Value  ::= var | type | Descriptions
```

Unsurprisingly, `type` and `attr` stand for a type and an attribute from the signature, while `var` is an XMG variable. Borrowing the notation from Definition 7, `attr:type` is an attribute-value pair. Figure 11 then shows the description language in action, while mimicking the AVM representation of the frame. Note that the order of descriptions within a pair of brackets is generally unrestricted, as well as the number of type expressions (in order to fully support conjunctive types). The specification language furthermore follows the definitions in Section 4 in that it allows for the abbreviated specification of paths, namely as a sequence of attributes separated by whitespaces. Hence, `[cause:[actor:?1]]` and `[cause actor:?1]` have the same meaning. Path equations are similarly constructed using the equality sym-

[24] In the following we use a simplified Backus-Naur form to define the syntax of specification languages. Disjunction (|), optionality (?) and the Kleene operators (*,+) are encoded as usual.

Figure 11: Specification of the frame component of n0Vn1pp(dir)

$$\boxed{0}\begin{bmatrix} \textit{causation} & \\ \text{ACTOR} & \boxed{1} \\ \text{THEME} & \boxed{2} \\ \text{CAUSE} & \begin{bmatrix} \textit{activity} & \\ \text{ACTOR} & \boxed{1} \\ \text{THEME} & \boxed{2} \end{bmatrix} \\ \text{EFFECT} & \boxed{4}\begin{bmatrix} \text{MOVER} & \boxed{2} \\ \text{GOAL} & \boxed{3} \end{bmatrix} \end{bmatrix}$$

```
<frame>{
?0[causation,
    actor:?1,
    theme:?2,
    cause:[activity,
           actor:?1,
           theme:?2],
    effect:?4[mover:?2,
              goal:?3]
]}
```

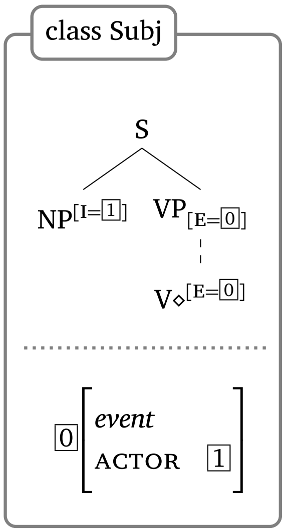
bol =. For example, the meaning of `[actor:?1,cause actor:?1]` could also be expressed with `[actor=cause actor:?1]`.[25]

The use of XMG variables not only enables one to avoid path equations, and eventually to mimic AVM representations, as we just demonstrated, but they may also serve to link semantic components with positions in the syntactic tree, as can be seen from Figure 12, where the ACTOR role and the NP-slot of the subject are linked in this way. It shows that XMG variables may play the role of base labels in labelled feature structure descriptions (see Section 4).

Finally, it might be instructive to compare the specification language for feature structure descriptions proposed here with alternatives that are used elsewhere, particularly in grammar development tools for HPSG. The ALE/TRALE system (Götz *et al.* 1997; Carpenter *et al.* 2003), for example, is rather similar in this respect. The only major difference is found in the path specification, where attributes are separated with the colon, which happens to be also the attribute-value separator. This overloading might be disadvantageous – and in fact path equations then entail a different way of specifying paths.[26]

[25] Note that, in the <syn>-dimension, the symbol = has a different meaning for historical reasons; there it acts as a attribute-value separator.

[26] Another case of overloading can be observed in PATR-II (Shieber 1984), where the operator for path equations and attribute-value pairs coincides. As a consequence, paths are enclosed by angled brackets to distinguish them from regular (atomic) values. Nevertheless, the notation of feature structures in Shieber (1984) bears similarity to our specification language. Indeed, one could say that we basically merge, and extend, the path and feature structure representations from PATR-II. See also the PC-PATR manual (McConnel 1995).



```
class Subj
...
<syn>{
  node ?S [cat=s];
  node ?SUBJ [cat=np,
              top=[i=?1]];
  node ?VP [cat=vp,bot=[e=?0]];
  node ?V (mark=anchor)
          [cat=v,top=[e=?0]];
  ?S->?SUBJ; ?S->?VP; ?VP->*?V;
  ?SUBJ>>?VP
}
<frame>{
?0[event,
    actor:?1]
}
...
```

Figure 12: Implementation of class Subj

In LKB (Copestake 2002), on the other hand, paths are encoded with dots instead of colons, while the whitespace acts as the separator in attribute-value pairs. This being merely an alphabetical variant of our proposal, there are other differences, notably the types being placed outside the feature structure brackets. Still, these differences seem rather marginal.

6.2.3 Specification language for feature structure constraints

We have seen in Section 4 that feature structure constraints can be – and in practice are – specified in different ways, namely either on the basis of a set of single constraint statements, or on the basis of a connected type hierarchy. Therefore one important aspect of the specification language for feature structure constraints is its versatility. Instead of dictating what direction to follow, the grammar writer should have several options at hand from which a suitable one may be chosen on a case-by-case basis.

Concerning the specification of feature structure constraints, two options are provided: a loose set of constraint statements, or a type hierarchy. The former are collected in the global field `frame_constraints`:

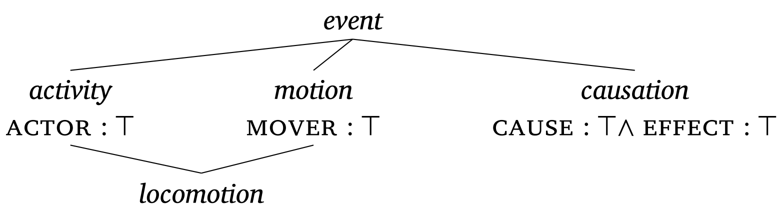
```
frame_constraints = {Constraint,Constraint,...}
Constraint ::=
  %% type constraint
  type+ '->' type+ |
  %% appropriateness condition
  type+ '->' Descriptions+  |
  %% feature-value constraint
  ('[' (AVPair|PathEquation) (',' AVPair|',' PathEquation)* ']')+
    '->' Descriptions+
Descriptions ::= '[' Description (',' Description)* ']'
Description  ::= type | PathEquation | AVPair
PathEquation ::= attr+ '=' attr+ (':' Value)?
AVPair ::= attr+ ':' Value
Value  ::= type | Descriptions
```

The specification language for constraints largely integrates the specification language for (unlabelled) descriptions, while it adds `->` as a symbol for generalized implication ($\preceq$ in Definition 7). Similarly, a distinction is made between type constraints, appropriateness conditions and feature-value constraints, to which we confine ourselves in the following. An example is provided in Figure 13. Note that the antecedent and the consequent of `->` may consist of more than one description separated by whitespaces, in which case the descriptions form a Cartesian product in the following way: t_1 t_2 `->` t_3 t_4 iff t_1 `->` t_3, t_1 `->` t_4, t_2 `->` t_3, t_2 `->` t_4. Furthermore it is possible to reverse `->`, hence to use t_2 `<-` t_1 instead of t_1 `->` t_2.

The other option is to specify feature structure constraints in the shape of a connected type hierarchy. For this the field `frame_type_hierarchy` is available and should be used in the following way:

```
frame_type_hierarchy = {Hierarchy,Hierarchy,...}
Hierarchy ::= '[' type (',' (Description|Hierarchy))* ']'
Descriptions ::= '[' Description (',' Description)* ']'
Description  ::= type | PathEquation | AVPair
PathEquation ::= attr+ '=' attr+ (':' Value)?
AVPair ::= attr+ ':' Value
Value  ::= type | Descriptions
```

An example of this way of specifying a type hierarchy was already included in Figure 13. Similarly to `frame_constraints`, only unlabelled feature structure descriptions are admitted, but unlike `frame_constraints`, the implication symbol `->` is missing. Instead squared



```
frame_constraints = {
    activity -> event, activity -> [actor:+],
    motion -> event, motion -> [mover:+],
    causation -> event, causation -> [cause:+,effect:+],
    locomotion -> activity motion}

frame_type_hierarchy = {
    [event, [activity, actor:+, [locomotion]],
            [motion, mover:+, [locomotion]],
            [causation, cause:+, effect:+]]}
```

Figure 13: Examples of the specification of feature structure constraints. The type hierarchy is a fraction of the type hierarchy shown above in Figure 3 from Kallmeyer and Osswald (2013)

brackets receive a second interpretation where they correspond to types. In other words, the set of feature structure constraints enclosed by a pair of brackets constitute the type constraints and appropriateness conditions for exactly one type. If there are several type expressions, then they form a conjoined type. And if another pair of type-denoting brackets is embedded, then this is a subtype of the embedding type. This squared bracket notation is introduced in the `Hierarchy` part of the syntax definition of `frame_type_hierarchy`, and it is also the only one that is used in the example in Figure 13.

Note here that `frame_type_hierarchy` can only express a proper subset of feature structure constraints, since constraints with feature structures in their antecedents are excluded from this representational format in general. Fortunately, constraints like these may be specified in parallel in the `frame_constraints` field.

Finally, there is the possibility to set a flag to trigger either the maximal or the minimal model of feature structure constraints:

```
use hierarchy (maximal|minimal) with dims (frame)
```

At the time of writing, minimal models are compiled by default.

6.3 *Compilation*

Within XMG2, the compilation of feature structure descriptions and constraints leads to representations which are used in the Prolog component of the system. Therefore we will be mainly concerned with Prolog data structures in the following. Note that, other than with tree descriptions in the <syn>-dimension, no underspecification is involved, and therefore the compiler for feature structure descriptions and constraints employs no constraint solving in the proper sense.

6.3.1 Compilation of feature structure descriptions

Typed feature structures are decomposed into two parts during compilation: types are compiled separately from feature structures. For the latter, we can reuse the XMG module (or brick) for untyped feature structures, which is already applied to feature structures in the <syn>-dimension. The module falls back on *attributed variables* and *association lists*, which are predefined data structures in Prolog. Attributed variables behave like common variables, the only difference being that they come with encapsulated attributes that can be accessed with dedicated commands only, and for which the standard unification algorithm can be customized. They are basically used here in order to replace the standard list (or term) unification by type unification and set union. Association lists, on the other hand, consist of key-value pairs with unique keys, thereby supporting the functionality of features within feature structures.

When a typed feature structure is declared, an attributed variable with two attributes is created: one attribute for the type and one for the feature-value pairs. Types are represented by bit vectors, basically Prolog lists over $\{0, 1\}$, that get unified by means of element-wise Boolean operations (see the next section). A set of feature-value pairs, on the other hand, is represented by an association list, whose values can be attributed variables again and thus induce recursion, i.e. feature structures of arbitrary depth. When two declared feature structures are unified, the compiler unifies their attributed variables in two different ways, namely with type unification on their type attribute and with set union on their association lists.

The check on the well-formedness of the generated structures is performed dynamically during their creation. This includes checking the appropriateness conditions (i.e. for invalid features or specific types of feature values) and computing type unifications. Contrary to the `<syn>`-dimension, the `<frame>`-dimension does not come with a description solver.

When the descriptions in the frame dimension are compiled, a number of instantiations and unifications of attributed variables is executed. It is important to note that the unification has to be explicitly specified by means of variable equations. Hence the compilation result may consist of several unconnected feature structures, as the compiler does not search for a minimal connected model that satisfies the processed feature structure descriptions. Furthermore note that, as unification is deterministic, there is at most one model for each accumulation.

6.3.2 Compilation of feature structure constraints

We are dealing with conjunctive types in the sense of Definition 8. Following a widespread approach (Aït-Kaci *et al.* 1989; Penn 1999; Kilbury *et al.* 2006; Skala *et al.* 2010), conjunctive types are internally represented by bit vectors (or "bit strings"), more precisely by Prolog lists over $\{0, 1\}$. The length of these lists is at least the number of elementary types in the signature. Every position in a bit vector stands for the membership of an elementary type in the conjunctive type, which means that a bit vector composed only of zeros is the most generic type (the empty set) and a bit vector composed only of ones is the conjunction of all elementary types.

We will present two methods to determine the set of valid bit vectors for a set of type constraints. The first one is a brute-force method that basically applies top-down filtering. The second one is based on subsumption matrices (Aït-Kaci *et al.* 1989; Penn 1999) and turns out to be more efficient. In the following let $T = \{a, b, c, d\}$ be the set of elementary types and # a bijective positioning function for bit vectors with $\# = \{(a, 1), (b, 2), (c, 3), (d, 4)\}$.

The top-down filtering on bit vector representations proceeds in the following way:

1. Generate the set of *virtual* (conjunctive) types, namely the bit vectors that represent the powerset of the set of elementary types:

$\{[1,0,0,0],[1,1,0,0],\ldots\}$.

2. Translate type constraints into bit vector patterns of *nonvalid* types:
$$\begin{aligned} a \preceq b &\quad\rightsquigarrow\quad [1,0,\_,\_] \\ a \wedge c \preceq \bot &\quad\rightsquigarrow\quad [1,\_,1,\_] \\ a \wedge b \preceq d &\quad\rightsquigarrow\quad [1,1,\_,0] \end{aligned}$$

3. Maximal model: Filter the set of bit vectors (representing the virtual types) based on the bit vector patterns in order to identify the valid types. Consequently, if no constraint is expressed, the set of valid types is the powerset of elementary types.

To compute the minimal model it needs an extra step:

3′. Minimal model: Translate elementary types and conjunctive types (on the left side of type constraints) into bit vector patterns of *declared* types:
$$\begin{aligned} a &\quad\rightsquigarrow\quad [1,\_,\_,\_] \\ a \wedge b \preceq d &\quad\rightsquigarrow\quad [1,1,\_,\_] \end{aligned}$$

Then, for each declared type, determine the set of compatible bit vectors from the maximal model and select the bit vector with the fewest 1s. If there is more than one such bit vector, compute the bitwise AND over all these bit vectors and add the resulting bit vector to the set of declared types.

4. Assign bit vector patterns to elementary types: $a \rightarrow [1,\_,\_,\_]$, $b \rightarrow [\_,1,\_,\_]$, …
5. Based on the set of valid bit vectors, unification of two types proceeds as list unification, after which the set of valid types is filtered with the resulting bit vector pattern. Finally the matching bit vector with the fewest 1s gets selected.

It is not hard to see that this method gets intractable for larger sets of elementary types, since the set of virtual types grows exponentially, namely with 2^n where n is the number of elementary types.

Fortunately, methods based on subsumption matrices tend to be much more space-efficient, although they still come with at least quadratic growth in terms of the number of elementary types. In the following, we adapt the procedure of Aït-Kaci *et al.* (1989) (see also Penn 1999), and only add to it anonymous types and maximal models:

1. Generate a boolean matrix where rows and columns correspond to elementary types and anonymous types that are subject to type

constraints. An element a_{ij} in the ith row and the jth column has value 1 iff $t_i \preceq t_j$ is a valid type constraint.[27] Otherwise it has value 0. For example, given type constraints $d \preceq c$ and $a \wedge b \preceq d$, the following preliminary matrix is generated:

$$\begin{array}{c|ccccc} & a & b & c & d & a \wedge b \\ \hline a & 1 & 0 & 0 & 0 & 1 \\ b & 0 & 1 & 0 & 0 & 1 \\ c & 0 & 0 & 1 & 1 & 0 \\ d & 0 & 0 & 0 & 1 & 1 \\ a \wedge b & 0 & 0 & 0 & 0 & 1 \end{array}$$

2. Multiply the matrix by itself until a fixpoint is reached.[28] This ensures that transitive subsumption relations are taken into account. When applied to the preliminary matrix above, we receive the following matrix, where c furthermore subsumes $a \wedge b$:

$$\begin{array}{c|ccccc} & a & b & c & d & a \wedge b \\ \hline a & 1 & 0 & 0 & 0 & 1 \\ b & 0 & 1 & 0 & 0 & 1 \\ c & 0 & 0 & 1 & 1 & 1 \\ d & 0 & 0 & 0 & 1 & 1 \\ a \wedge b & 0 & 0 & 0 & 0 & 1 \end{array}$$

3. Each type is assigned two vectors: a subsumption vector taken from the row, and an ISA vector taken from the column (which was also used in the previous filtering approach). Hence the bit vector representation for type a consists of the subsumption vector $[1,0,0,0,1]$ and the ISA vector $[1,0,0,0,0]$. The difference of the minimal and maximal model emerges in the use of these vector pairs.
4. Minimal model: During unification the subsumption vectors are combined with bitwise AND in order to determine the most general common subtype (Aït-Kaci *et al.* 1989). To give an example, the unification of the subsumption vectors assigned to a and d results in the subsumption vector of $a \wedge b$:

[27] We tacitly assume that tautologies (e.g. $t_i \preceq t_i$ and $t_i \wedge t_j \preceq t_i$) are taken into account.

[28] According to Aït-Kaci *et al.* (1989), this takes at most $log_2 n$ iterations.

$$\begin{array}{cc} a & [1,0,0,0,1] \\ d & [0,0,0,1,1] \\ \hline a \wedge b & [0,0,0,0,1] \end{array}$$

Otherwise, if the resulting bit vector was not found among the bit vectors of the matrix, the unification does not necessarily fail. It only fails if the resulting bit vector only consists of 0s.

4′. Maximal model: During unification, the ISA vectors are combined with bitwise OR, while the subsumption vectors are ignored. Hence the unification of a and d yields an ISA vector not found in the matrix:

$$\begin{array}{cc} a & [1,0,0,0,0] \\ d & [0,0,1,1,0] \\ \hline (a \wedge d) & [1,0,1,1,0] \end{array}$$

In order to account for constraints on anonymous types, further unification steps can be necessary. For example, upon unifying the ISA vectors of a and b, the ISA vector of $a \wedge b$ has to be added as well:

$$\begin{array}{cc} a & [1,0,0,0,0] \\ b & [0,1,0,0,0] \\ \hline & [1,1,0,0,0] \\ a \wedge b & [1,1,1,1,1] \\ \hline & [1,1,1,1,1] \end{array}$$

Types that are explicitly ruled out in the type constraints are accounted for by means of extra filtering. For example, a type constraint such as $a \wedge d \preceq \bot$ would give rise to a filter based on the pattern $[1,\_,\_,1,\_]$, which would be applied to the resulting ISA vector after unification.

For each one of the computed valid types, i.e. the assigned ISA vectors, XMG also computes the set of appropriateness conditions. Once again, this step uses the technique of vector pattern matching: when a valid type matches a pattern corresponding to a appropriateness condition, the appropriateness condition is added to the list of appropriateness conditions for this type. These lists can be used in other precompilation steps to check for cyclicity in the feature structure constraints, and for their incompatibility.

At the end of the compilation process, the bit vectors can be easily mapped back to conjunctive types and the corresponding elementary type, if it exists.

7 DISCUSSION AND CONCLUSION

In this article, we presented recent efforts to extend the grammar engineering framework XMG in order to deal with frame representations in the format of typed feature structures. Because metagrammatical factorization involves the composition of feature structures and types along given feature structure constraints, the full power of unification on typed feature structures is needed in XMG. We showed that the simulation of typed feature structures within the `<sem>`-dimension comes with severe disadvantages concerning the implementation of types and type unification. Therefore a new toolkit was developed, including a novel `<frame>`-dimension, which is adjusted to the peculiarities of typed feature structures and type unification, and which should eventually reduce the burden for the grammar writer. The article explained the main components of this toolkit: the specification language, for which a comprehensive code example is included in the appendix, and the compilation procedure, which uses bit vector encodings of types.

While the focus was on the theoretic foundation of the proposed extension and its proof of concept, aspects of computational complexity and the possibilities for optimization (e.g. regarding the size of bit vectors) were largely set aside. Undoubtedly, there is room for future improvements of the compiler. It has to be stressed, however, that compilation with XMG is not as time-critical as parsing, because XMG compilation is part of the preprocessing (Kallmeyer and Osswald 2013, 56). Nevertheless it would be interesting to see how compilation time scales as a function of the size and structure of the type hierarchy, and whether this is relevant given theoretically justified conceptual type systems. As no complex examples of the latter kind are known to us, this remains to be seen.[29] We are aware, of course, that the issue of complexity will become more critical once these resources are used for parsing. Note that, while the presented extensions to XMG are fully

[29] Just for comparison, Skala and Penn (2011) count some 3412 elementary types in the English Resource Grammar (ERG, Copestake and Flickinger 2000).

operational in a recent prototype, a compatible lexical component as well as a parser have yet to be implemented.

Regardless of complexity issues, it could be useful to separate the sources and compilation procedures for global feature structure constraints on the one hand, and local feature structure descriptions within the `<frame>`-dimension on the other hand. The reason is that at some point the feature structure constraints should stabilize, given that they represent cognitive concepts, and therefore constant recompilation triggered by changes in other parts of the metagrammar would be superfluous. At the current stage of research, however, as the development of frame representations is still ongoing, we think that metagrammars are the right place to implement and to experiment with feature structure constraints.

Finally it remains to be stressed that the combination of the `<frame>`-dimension with the `<syn>`-dimension is by no means privileged. The `<frame>`-dimension can also be used to implement standalone frames, or to implement recent frame-based accounts to morphological decomposition (e.g. Zinova and Kallmeyer 2012), thereby considerably widening the scope of XMG.

APPENDIX: COMPLETE CODE EXAMPLE

The following code example covers the implementation of the type hierarchy in Figure 3 and the factorization of the prepositional object construction in Figures 4 and 5.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HEADER:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
type MARK = {subst, nadj, foot, anchor, coanchor, flex, lex}
type CAT = {np,v,vp,s,pp}
type VAR !
property mark : MARK
feature cat : CAT
feature top : VAR
feature bot : VAR
feature i : VAR
feature e : VAR
feature path : VAR

frame_types = {event,activity,motion,causation,translocation,
    onset-causation,extended-causation,locomotion,bounded-
    translocation,bounded-locomotion}
frame_attributes = {actor,theme,goal,mover,path,cause,effect}
frame_constraints = { activity -> event,
  activity -> actor:+,
  motion -> event,
  motion -> mover:+,
  causation -> event,
  causation -> cause:+,
  causation -> effect:+,
  [activity,motion] -> actor=mover,
  translocation -> motion,
  translocation -> path:+,
  bounded-translocation -> translocation,
  bounded-translocation -> goal:+,
  locomotion -> activity translocation,
  bounded-locomotion -> locomotion bounded-translocation
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TREE FRAGMENTS:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class VSpine
export ?V
declare  ?VP ?V ?X0
{<syn>{
  node ?VP [cat=vp] {
      node ?V (mark=anchor)[cat=v]
} } }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class Subj
export ?V ?X0
declare ?S ?NP ?VP ?V ?X0 ?X1
{ <syn>{
    node ?S [cat = s];
    node ?NP (mark=subst)[cat=np,top=[i=?X1]];
    node ?VP [cat=vp,bot=[e=?X0]];
    node ?V (mark=anchor)[cat=v,top=[e=?X0]];
    ?S -> ?NP; ?S -> ?VP; ?VP ->* ?V; ?NP >> ?VP
  };
  <frame> {
  ?X0[event,
    actor:?X1]
} }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class DirObj
export ?V ?X0
declare ?VP ?NP ?V ?X0 ?X2
{ <syn>{
  node ?VP [cat = vp,bot=[e=?X0]]{
    node ?V [cat=v,top=[e=?X0] ]
    ,,,node ?NP (mark=subst)[cat=np,top=[i=?X2] ]
  } };
  <frame>{
  ?X0[event,goal:?X2]
  |
  ?X0[event,theme:?X2]
} }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class DirPrepObj
export ?V ?X0
declare ?VP1 ?VP2 ?PP ?V ?X0 ?X1 ?X2 ?X3
{ <syn>{
  node ?VP1 [cat = vp,bot=[path=?X3] ]{
    node ?VP2 [cat = vp,bot=[path=?X3] ]{
      node ?V (mark=anchor)[cat=v]}
    node ?PP (mark=subst)[cat=pp,top=[i=?X1,e=?X0] ]
  } };
  <frame>{
  ?X0[bounded-translocation,
    goal:?X1,
    path:?X3]
} }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TREE TEMPLATES:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class n0V
export ?V ?X0
declare ?V ?SSubj ?Spine ?X0
{
  ?SSubj = Subj[];
  ?Spine = VSpine[];
  ?SSubj.?V = ?V;
  ?Spine.?V = ?V;
  ?SSubj.?X0 =?X0
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class n0Vn1
import n0V[]
declare ?Obj
{
  ?Obj = DirObj[];
  ?Obj.?V = ?V
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
class n0Vn1pp-dir
import n0Vn1[]
declare ?PPObj ?X1 ?X2 ?X3 ?X4
{
  ?PPObj = DirPrepObj[];
  ?PPObj.?V = ?V;
  ?PPObj.?X0 = ?X4;
  <frame>{
  ?X0[causation,
    actor:?X1,
    theme:?X2,
    cause:[activity,
        actor:?X1,
        theme:?X2],
    effect:?X4[
      mover:?X2,
      goal:?X3]
  ]
} }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EVALUATION:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
value n0V
value n0Vn1
value n0Vn1pp-dir
```

REFERENCES

Anne ABEILLÉ and Owen RAMBOW (2000a), Tree Adjoining Grammar: An overview, in Abeillé and Rambow (2000b), pp. 1–68.

Anne ABEILLÉ and Owen RAMBOW, editors (2000b), *Tree Adjoining Grammars: Formalisms, linguistic analyses and processing*, number 107 in CSLI Lecture Notes, CSLI Publications, Stanford, CA.

Hassan AÏT-KACI, Robert BOYER, Patrick LINCOLN, and Roger NASR (1989), Efficient implementation of lattice operations, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):115–146.

Lawrence BARSALOU (1992), Frames, concepts, and conceptual fields, in Adrienne LEHRER and Eva Feder KITTEY, editors, *Frames, fields, and contrasts: New essays in semantic and lexical organization*, pp. 21–74, Lawrence Erlbaum Associates, Hillsdale, NJ.

Tilman BECKER (1994), *HyTAG: A new type of Tree Adjoining Grammars for hybrid syntactic representations of free word order languages*, Ph.D. thesis, Universität des Saarlandes,
`http://www.dfki.de/~becker/becker.diss.ps.gz`.

Tilman BECKER (2000), Patterns in metarules for TAG, in Abeillé and Rambow (2000b), pp. 331–342.

Johan BOS (1996), Predicate logic unplugged, in Paul DEKKER and Martin STOKHOF, editors, *Proceedings of the tenth Amsterdam Colloquium*, pp. 133–143, Amsterdam, Netherlands.

Marie-Hélène CANDITO (1996), A principle-based hierarchical representation of LTAGs, in *Proceedings of the 16th International Conference on Computational Linguistics (COLING 96)*, Copenhagen, Denmark,
`http://aclweb.org/anthology-new/C/C96/C96-1034.pdf`.

Bob CARPENTER (1992), *The logic of typed feature structures with applications to unification grammars, logic programs and constraint resolution*, number 32 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, UK.

Bob CARPENTER, Gerald PENN, and Mohammad HAJI-ABDOLHOSSEINI (2003), *The Attribute Logic Engine user's guide with TRALE extensions*, version 4.0 beta edition.

Bob CARPENTER and Carl POLLARD (1991), Inclusion, disjointness and choice: The logic of linguistic classification, in *Proceedings of the 29th annual meeting of the Association for Computational Linguistics*, pp. 9–16, Berkeley, CA,
`http://acl.ldc.upenn.edu/P/P91/P91-1002.pdf`.

Ann COPESTAKE (2002), *Implementing typed feature structure grammars*, CSLI Publications, Stanford, CA.

Ann A. COPESTAKE and Dan FLICKINGER (2000), An open source grammar development environment and broad-coverage English grammar using HPSG, in *Proceedings of the second Conference on Language Resources and Evaluation (LREC 2000)*, Athens, Greece.

Benoit CRABBÉ and Denys DUCHIER (2005), Metagrammar redux, in Henning CHRISTIANSEN, Peter Rossen SKADHAUGE, and Jørgen VILLADSEN, editors, *Constraint solving and language processing*, number 3438 in Lecture Notes in Computer Science, pp. 32–47, Springer, Berlin, Germany.

Benoit CRABBÉ, Denys DUCHIER, Claire GARDENT, Joseph LE ROUX, and Yannick PARMENTIER (2013), XMG: eXtensible MetaGrammar, *Computational Linguistics*, 39(3):1–66,
`http://hal.archives-ouvertes.fr/hal-00768224/en/`.

Denys DUCHIER, Brunelle Magnana EKOUKOU, Yannick PARMENTIER, Simon PETITJEAN, and Emmanuel SCHANG (2012), Describing morphologically rich languages using metagrammars: A look at verbs in Ikota, in *Workshop on Language Technology for Normalisation of Less-Resourced Languages (SALTMIL 8 - AfLaT 2012)*, pp. 55–59, Istanbul, Turkey, `http://www.tshwanedje.com/publications/SaLTMiL8-AfLaT2012.pdf#page=67`.

Charles J. FILLMORE (1982), Frame Semantics, in The Linguistic Society of Korea, editor, *Linguistics in the morning calm*, pp. 111–137, Hanshin Publishing, Seoul, South Korea.

Charles J. FILLMORE (2007), Valency issues in FrameNet, in Thomas HERBST and Katrin GÖTZ-VOTTELER, editors, *Valency: Theoretical, descriptive and cognitive issues*, pp. 129–162, Mouton de Gruyter, Berlin, Germany.

Dan FLICKINGER (2000), On building a more effcient grammar by exploiting types, *Natural Language Engineering*, 6(1):15–28.

Anette FRANK and Josef VAN GENABITH (2001), GlueTag. Linear Logic based Semantics for LTAG – and what it teaches us about LFG and LTAG, in Miriam BUTT and Tracy Holloway KING, editors, *Proceedings of the LFG01 conference*, CSLI Publications, Hong Kong.

Claire GARDENT and Laura KALLMEYER (2003), Semantic construction in Feature-Based TAG, in *Proceedings of the 10th meeting of the European Chapter of the Association for Computational Linguistics*, pp. 123–130.

Adele GOLDBERG (2006), *Constructions at work. The nature of generalizations in language*, Oxford University Press, Oxford, UK.

Thilo GÖTZ, Detmar MEURERS, and Dale GERDEMANN (1997), *The ConTroll manual*, Seminar für Sprachwissenschaft, Universität Tübingen, Tübingen, Germany,
`http://www.sfs.uni-tuebingen.de/controll/controll-manual.ps.gz`, draft of 17. September 1997 for ConTroll v.1.0b and XTroll v.5.0b.

Aravind K. JOSHI and Yves SCHABES (1997), Tree-Adjoining Grammars, in Grzegorz ROZENBERG and Arto SALOMAA, editors, *Handbook of Formal Languages*, volume 3, pp. 69–124, Springer, Berlin, Germany.

Aravind K. JOSHI, K.Vijay SHANKER, and David WEIR (1990), The convergence of mildly context-sensitive grammar formalisms, Technical Report MS-CIS-90-01, Department of Computer and Information Science, University of Pennsylvania, `http://repository.upenn.edu/cis_reports/539/`.

Laura KALLMEYER, Timm LICHTE, Wolfgang MAIER, Yannick PARMENTIER, Johannes DELLERT, and Kilian EVANG (2008), TuLiPA: Towards a multi-formalism parsing environment for grammar engineering, in *Proceedings of the workshop on Grammar Engineering Across Frameworks (GEAF '08)*, pp. 1–8, Manchester, UK.

Laura KALLMEYER and Rainer OSSWALD (2012a), An analysis of directed motion expressions with Lexicalized Tree Adjoining Grammars and frame semantics, in Luke ONG and Ruy DE QUEIROZ, editors, *Proceedings of WoLLIC*, number 7456 in Lecture Notes in Computer Science (LNCS), pp. 34–55, Springer, Berlin, Germany.

Laura KALLMEYER and Rainer OSSWALD (2012b), A frame-based semantics of the dative alternation in Lexicalized Tree Adjoining Grammars, in Christopher PIÑÓN, editor, *Empirical Issues in Syntax and Semantics 9*, pp. 167–184, Paris, France.

Laura KALLMEYER and Rainer OSSWALD (2013), Syntax-driven semantic frame composition in Lexicalized Tree Adjoining Grammar, *Journal of Language Modelling*, 1:267–330.

Paul KAY (2002), An informal sketch of a formal architecture for Construction Grammar, *Grammars*, 5:1–19.

James KILBURY, Wiebke PETERSEN, and Christof RUMPF (2006), Inheritance-based models of the lexicon, in Dieter WUNDERLICH, editor, *Advances in the theory of the lexicon*, number 13 in Interface Explorations, pp. 429–478, De Gruyter, Berlin, Germany.

Timm LICHTE, Alexander DIEZ, and Simon PETITJEAN (2013), Coupling trees, words and frames through XMG, in *Proceedings of the ESSLLI 2013 workshop on high-level methodologies for grammar engineering*, Düsseldorf, Germany.

Robert MALOUF, John CARROLL, and Ann COPESTAKE. (2000), Efficient feature structure operations without compilation, *Natural Language Engineering*, 6(1):29–46.

Stephen MCCONNEL (1995), *PC-PATR reference manual*, `http://www.sil.org/pcpatr/manual/pcpatr.html`, version 0.97a9.

Rainer OSSWALD (2003), *A logic of classification with applications to linguistic theory*, Dissertation, FernUniversität Hagen.

Rainer OSSWALD and Robert D. VAN VALIN, Jr. (2014), FrameNet, frame structure, and the syntax-semantics interface, in Thomas GAMERSCHLAG, Doris GERLAND, Rainer OSSWALD, and Wiebke PETERSEN, editors, *Frames and concept types*, number 94 in Studies in Linguistics and Philosophy, pp. 125–156, Springer, Cham, Switzerland.

Gerald PENN (1999), An optimized Prolog encoding of typed feature structures, Arbeitspapiere des SFB 340 138, University of Tübingen.

Guy PERRIER (2000), Interaction Grammars, in *Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*, pp. 600–606, Saarbrücken, Germany.

Wiebke PETERSEN (2007), Representation of concepts as frames, *The Baltic International Yearbook of Cognition, Logic and Communication*, 2:151–170.

Wiebke PETERSEN and Tanja OSSWALD (2014), Concept composition in frames: Focusing on genitive constructions, in Thomas GAMERSCHLAG, Doris GERLAND, Rainer OSSWALD, and Wiebke PETERSEN, editors, *Frames and concept types*, number 94 in Studies in Linguistics and Philosophy, pp. 243–266, Springer, Cham, Switzerland.

Carlos A. PROLO (2002), Generating the XTAG English grammar using metarules, in *Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002)*, pp. 814–820, Taipei. Taiwan.

Stuart M. SHIEBER (1984), The design of a computer language for linguistic information, in *Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting of the Association for Computational Linguistics*, pp. 362–366, Stanford, CA, http://www.aclweb.org/anthology/P84-1075.

Stuart M. SHIEBER and Yves SCHABES (1990), Synchronous Tree-Adjoining Grammars, in *Proceedings of the 13th International Conference on Computational Linguistics (COLING 1990)*, pp. 253–258, Helsinki, Finland.

Matthew SKALA, Victoria KRAKOVNA, János KRAMÁR, and Gerald PENN (2010), A generalized-zero-preserving method for compact encoding of concept lattices, in *Proceedings of the 48th annual meeting of the Association for Computational Linguistics*, pp. 1512–1521, Uppsala, Sweden.

Matthew SKALA and Gerald PENN (2011), Approximate bit vectors for fast unification, in Makoto KANAZAWA, András KORNAI, Marcus KRACHT, and Hiroyuki SEKI, editors, *The mathematics of language*, number 6878 in Lecture Notes in Computer Science, pp. 158–173, Springer, Berlin, Germany.

Matthew STONE and Christine DORAN (1997), Sentence planning as description using Tree Adjoining Grammar, in *Proceedings of the eighth conference of the European Chapter of the Association for Computational Linguistics (EACL'97)*, pp. 198–205.

Fei XIA, Martha PALMER, and K. VIJAY-SHANKER (2010), Developing Tree-Adjoining Grammars with lexical descriptions, in Srinivas BANGALORE and Aravind K. JOSHI, editors, *Using complex lexical descriptions in natural language processing*, pp. 73–110, MIT Press, Cambridge, UK.

Yulia ZINOVA and Laura KALLMEYER (2012), A frame-based semantics of locative alternation in LTAG, in *Proceedings of the 11th international workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11)*, pp. 28–36, Paris, France, http://www.aclweb.org/anthology-new/W/W12/W12-4604.