The XDG Grammar Development Kit

Ralph Debusmann[1], Denys Duchier[2], and Joachim Niehren[3]

[1] Saarland University, Programming Systems Lab, Saarbücken, Germany
[2] LORIA, Équipe Calligramme, Nancy, France
[3] INRIA Futurs, Mostrare Project, Lille, France

Abstract. Extensible Dependency Grammar (XDG) is a graph description language whose formulas can be solved by constraint programming. XDG is designed so as to yield a declarative approach to natural language processing, in particular to parsing and generation. In this paper, we present the XDG Development Kit (XDK), the first XDG-based grammar development system, which we have implemented in Mozart/Oz, thereby making full use of its multi-paradigmatic nature. The XDK supports an expressive lexicon specification language which which has not been published previously.

1 Introduction

Declarative grammar formalisms have a long tradition for modeling and processing natural language syntax and semantics [1, 2]. The idea is to specify linguistic knowledge in grammars independently from processing aspects, such that parsers, semantic constructions, or sentence generators can be created generically for all grammars of a given formalism.

The most prominent grammar formalisms support dialects of Lexical Functional Grammar (LFG) [1], Head-Driven Phrase Structure Grammar (HPSG) [3], Categorial Grammar [4, 5], Tree Adjoining Grammar (TAG) [6, 7], and Dependency Grammar (DG) [8, 9].

Grammar development systems are collections of tools that support the development of grammars in some formalism. They offer a concrete syntax for grammar specification, and contain parsers, generators, graphical output tools, debugging facilities, etc. The most powerful grammar development systems are the LKB system [10] for HPSG, the XTAG system [11] for TAG, and the Grammar Writer's Workbench [12] for LFG.

Parsers for grammars in LFG and HPSG rely on first-order unification for feature structures. Smolka raised the question [13] whether more advanced constraint technology could help to improve existing natural language processing methods. Duchier [14] proposed a first solution to this question. Motivated by Dependency Grammar, he proposed to axiomatize valid *dependency graphs* by finite set constraints, and reduced parsing to *finite set constraint programming*. Duchier and Debusmann [15] developed this approach further into a grammar formalism called Topological Dependency Grammar (TDG), which is particularly well suited for free word order, as in German, Czech, Latin, etc.

Recently, Debusmann et. al. [16] proposed a further generalization, Extensible Dependency Grammar (XDG). This is a general graph description language flexible enough to model multiple levels of linguistic structure, while still enjoying the same constraint-based parsing techniques [17]. In particular, XDG permits to extend TDG by a constraint-based, bi-directional syntax-semantics interface.

In this paper, we propose the first grammar development system for XDG, the XDG Grammar Development Kit (XDK). This includes a lexicon description language that has not been published previously. We have implemented the XDK in Mozart/Oz and published it in the MOGUL library [18]. The XDK provides a comprehensive suite of facilities for grammar development. It offers multiple concrete syntaxes for grammar specification: one XML-based for automatic grammar creation, and one more human-friendly for handcrafted grammars. Moreover, it provides a solver for parsing and generation, and various graphical output tools and debugging facilities. All of this is implemented in one coherent system, making use of the multi-paradigmatic nature of Mozart/Oz: We could use object-oriented programming for the GUI, functional programming for grammar compilation, and constraint programming for the solver.

2 Extensible Dependency Grammar

In XDG, we regard grammars as graph descriptions. This allows us to view parsing of natural language expressions and the generation of sentences as graph configuration problems which can be solved using constraint programming in Mozart/Oz.

2.1 Graphs

XDG describes finite labeled graphs, using the linguistic notion of Dependency Grammar [8, 9]. We show a typical *dependency graph* in Fig. 1 (left hand side). Each node corresponds one-to-one to a word in the sentence. The edges are labeled by grammatical relations such as subject, object and determiner. Here, "programmer" is the subject of "should", and "like" the verbal complement (vcomp). "every" is the determiner of "programmer", and "Mozart" is the object of "like".

An XDG analysis can be split up into an arbitrary number of dependency graphs, all sharing the same set of nodes, but having different edges. This is useful for the handling of word order [15], and for the representation of the semantics of natural language. We call each of the graphs a *dimension*. In Fig. 1 (right), we display an analysis of the same sentence on a second, semantic dimension. Here, "programmer" is simultaneously the agent of "should", and the agent of "like". "every" is the determiner of "programmer". "like" is the proposition of "should", and "Mozart" the patient of "like".

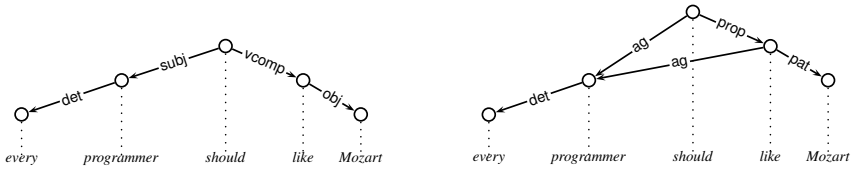

Fig. 1. Syntactic (left) and semantic (right) dependency graphs

2.2 Graph Description Language

XDG describes the well-formedness conditions of an analysis by the interaction of *principles* and the *lexicon*. The principles stipulate restrictions on one or more of the dimensions, and are controlled by the feature structures assigned to the nodes from the lexicon. Here is a lexical entry for "like":

$$\text{“like”} = \begin{bmatrix} \mathit{syn}: \begin{bmatrix} \mathsf{in}: \{\mathsf{vcomp?}\} \\ \mathsf{out}: \{\mathsf{obj!}\} \end{bmatrix} \\ \mathit{sem}: \begin{bmatrix} \mathsf{in}: \{\mathsf{prop?}\} \\ \mathsf{out}: \{\mathsf{ag!}, \mathsf{pat!}\} \end{bmatrix} \end{bmatrix}$$

The entry is separated into a syntactic and a semantic part, and controls the *valency principle*, constraining the licensed incoming and outgoing edges of each node. In the syntax, "like" can have zero or one incoming edges labeled vcomp (in : {vcomp?}), and requires an object (out : {obj!}). In the semantics, it can have zero or one incoming edges labeled prop (in : {prop?}) and requires an agent and a patient (out : {ag!, pat!}).

XDG is "extensible" for two reasons: 1) the set of dimensions of graphs is arbitrary, and 2) the set of principles to describe the graphs is a subset of an extensible *principle library*. The principle library already contains the necessary principles to model the syntax and semantics for large fragments of German and English, and smaller fragments of Arabic, Czech and Dutch. We present a subset of the principle library below.

Tree principle. Dimension i must be a tree. In the example above, we use this principle on the syntactic dimension.

DAG principle. Dimension i must be a directed acyclic graph. We use this principle on the semantic dimension.

Valency principle. For each node on dimension i, the incoming edges must be licensed by the in specification, and the outgoing edges by the out specification.

Order principle. For each node v on dimension i, the order of the daughters depends on their edge labels. We use this principle to constrain the order of the words in a sentence. We can use it e.g. to require that subjects ("programmer") precede verbal complements ("like").

Linking principle. The linking principle allows us to specify how semantic arguments must be realized in the syntax. In our example, the lexical entry for "like" would contain the following feature specification:

$$\text{“like”} = \left[\mathit{sem} : \left[\mathsf{link} : \left[\begin{array}{l} \mathsf{ag} : \{\mathsf{subj}\} \\ \mathsf{pat} : \{\mathsf{obj}\} \end{array} \right] \right] \right]$$

This stipulates that the agent of "like" must be realized by the subject, and the patient by the object.

3 Lexicon Specification

The XDG development kit offers a flexible method to define types of lexical entries, to build lexical abstractions, and to describe sets of lexical entries compactly using a descriptive device known as *metagrammar*. A metagrammar is processed to automatically generate all the entries of an XDG lexicon.

3.1 Lexicalization

Lexicalization is a widely accepted principle in computational linguistics that is indispensable in formal grammar approaches. Lexicalization means that linguistic information is mostly specified in the lexicon, given that information is often mostly specific to words.

The lexicon quickly becomes huge even for grammars with moderately ambitious coverage. They may contain thousands of words, each of which having multiple lexical entries, which are often large too. From the engineering perspective, it is important to provide facilities that allow to adequately modularize and factorize lexical information; otherwise, information needs to be duplicated and maintained in multiple places.

3.2 Ambiguity

XDG is very much a lexicalized grammar formalism. Most information is specified in the lexical entries for the words. The exceptions are some of the principles, which specify how words can interact, or how graphs in different dimensions are related.

We have already seen XDG lexical entries in the examples. Lexical entries are records of dimensions, and each dimension is itself a record representing linguistic information which pertains to the word. These items may have different types. We have already seen valency stipulations, specifying which labeled edges are permitted to enter or exit a node in a graph, and linking specifications, specifying how semantic arguments are to be realized in the syntax.

In a typical lexicon, there are a number of lexical entries for each word, e.g. if a word has different categories: "help" for instance can either be a noun or a verb. The problem is to describe such sets of lexical entries compactly, without representing the same information in different lexical entries twice. XDG provides lexical abstractions for this purpose.

3.3 Lexical Types

XDG supports a flexible system to define various types of lexical information. Each type consists of a set L and a partial function $\sqcap : L \times L \to L$, the *combination function* of L. Most typically, the operation $\sqcap$ defines the greatest lower bound with respect to the information amount represented by members of L.

The grammar writer starts by defining some domain types, for instance the type of edge labels in the syntactic dimension:

$$\mathit{syn.label} = \{\texttt{det}, \texttt{subj}, \texttt{obj}, \texttt{vcomp}\}$$

Domain types are always flat in that $a \sqcap a = a$ for all elements and $a \sqcap b$ is undefined for all $a \neq b$. Given a set of features $(\mathsf{f}_i)_{i=1\ldots n}$ and a corresponding set of types $T_i{=}(L_i, \sqcap_i)_{i=1\ldots n}$, XDG allows you to define the record type $[\mathsf{f}_1{:}T_1, \ldots, \mathsf{f}_n{:}T_n]$ with values of the form:

$$[\mathsf{f}_1{:}v_1, \ldots, \mathsf{f}_n{:}v_n]$$

where $v_i \in L_i$, and where the composition operation is defined feature-wise by:

$$[\mathsf{f}_1{:}v_1, \ldots, \mathsf{f}_n{:}v_n] \sqcap [\mathsf{f}_1{:}v'_1, \ldots, \mathsf{f}_n{:}v'_n] \quad = \quad [\mathsf{f}_1{:}v_1 \sqcap_1 v'_1, \ldots, \mathsf{f}_n{:}v_n \sqcap_n v'_n]$$

when $v_i \sqcap_i v'_i$ are all defined, and is undefined otherwise.

The grammar writer needs to define a type for valencies on the syntactic level. The XDG system provides a built-in constructor to define valencies over a given domain type of edge labels:

$$\mathit{syn.valency} = \texttt{valency}(\mathit{syn.label})$$

This merely defines *syn.valency* to be the record type:

$$[\texttt{det}{:}\mathit{mode},\ \texttt{subj}{:}\mathit{mode},\ \texttt{obj}{:}\mathit{mode},\ \texttt{vcomp}{:}\mathit{mode}]$$

where type *mode* consists of the values $\{0, ?, !, *\}$ — where 0 stands for no occurrence, ! for one unique and obligatory occurrence, ? for an optional occurrence, and $*$ for zero or more occurrences — and the following (commutative) combination operation:

$$0 \sqcap x = x \qquad * \sqcap ! = ! \qquad * \sqcap ? = ? \qquad ? \sqcap ! = !$$

Since *syn.valency* was declared with the `valency` constructor, the XDK supports the following more convenient notation:

$$\{\texttt{subj}!, \texttt{obj}?\} \quad \equiv \quad [\texttt{det}{:}0,\ \texttt{subj}{:}!,\ \texttt{obj}{:}?,\ \texttt{vcomp}{:}0]$$

In practice, record types serve for defining dimensions and lexical entries. A lexical entry is a record of named dimensions, and a dimension a record of lexical information about valency, agreement etc... The XDK also supports defining new types using cartesian products, set type constructors, and other possibilities.

3.4 Lexical Meta Grammars

Once we have specified the type $(L, \sqcap)$ of lexical entries, we need to supply the set of values of this type that constitute the lexicon. For this purpose, we adapt a well-known descriptive device: generative grammar, consisting of a finite set of clauses with the following abstract syntax:

$$\begin{aligned} \mathit{Clause} &::= \mathit{Name} \rightarrow \mathit{Goal} \\ \mathit{Goal} &::= \mathit{Goal} \wedge \mathit{Goal} \mid \mathit{Goal} \vee \mathit{Goal} \mid \mathit{Name} \mid c \end{aligned}$$

where each *Clause* defines a non-terminal *Name*, and where the terminals c range over elements of L, i.e. lexical entries Traditional context free grammars are similar. Name correspond to non-terminals and elements of $c \in L$ to terminals. Conjunction is usually written as juxtaposition, and disjunction as choice $|$. Here, we use grammars to describe sets of lexical entries. Compared to the traditional semantics, we replace words by lexical entries and word concatenation by the $\sqcap$ operator on lexical entries. We call such a device a *metagrammar* over $(L, \sqcap)$.

3.5 Example

In this section, we present a simple, idealized example of a metagrammar. First, we state that finite verbs can either be the head of the main clause or of a relative clause, i.e. either they have no incoming edges, or they can have incoming edge rel:

$$\begin{aligned} \mathsf{finite} &\rightarrow \mathsf{root} \vee \mathsf{rel} \\ \mathsf{root} &\rightarrow [\, \mathit{syn} : [\, \mathsf{in} : \{\}\,]\,] \\ \mathsf{rel} &\rightarrow [\, \mathit{syn} : [\, \mathsf{in} : \{\mathsf{relcl?}\}\,]\,] \end{aligned}$$

Then we state that verbs may be either intransitive, transitive or ditransitive:

$$\begin{aligned} \mathsf{verb} &\rightarrow \mathsf{intr} \vee \mathsf{tr} \vee \mathsf{ditr} \\ \mathsf{intr} &\rightarrow [\, \mathit{syn} : [\, \mathsf{out} : \{\mathsf{subj!}\}\,]\,] \\ \mathsf{tr} &\rightarrow \mathsf{intr} \wedge [\, \mathit{syn} : [\, \mathsf{out} : \{\mathsf{obj!}\}\,]\,] \\ \mathsf{ditr} &\rightarrow \mathsf{tr} \wedge [\, \mathit{syn} : [\, \mathsf{out} : \{\mathsf{iobj!}\}\,]\,] \end{aligned}$$

The notion of a finite verb can be stated as the composition of the previous two abstractions:

$$\mathsf{finite.verb} \rightarrow \mathsf{finite} \wedge \mathsf{verb}$$

The generative process using finite.verb as start symbol produces the following six values which are alternative lexical entries for finite verbs:

$$(\mathsf{root} \wedge \mathsf{intr}) \quad (\mathsf{root} \wedge \mathsf{tr}) \quad (\mathsf{root} \wedge \mathsf{ditr}) \quad (\mathsf{rel} \wedge \mathsf{intr}) \quad (\mathsf{rel} \wedge \mathsf{tr}) \quad (\mathsf{rel} \wedge \mathsf{ditr})$$

For instance, the lexical entry for a ditransitive finite verb which is the head of a relative clause is:

$$\mathsf{rel} \wedge \mathsf{ditr} \rightarrow \left[\, \mathit{syn} : \begin{bmatrix} \mathsf{in} : \{\mathsf{relcl?}\} \\ \mathsf{out} : \{\mathsf{subj!}, \mathsf{obj!}, \mathsf{iobj!}\} \end{bmatrix}\right] \tag{1}$$

4 XDG Grammar Development Kit

The XDK is a complete grammar development kit for XDG. It defines concrete syntaxes for grammar specification, and various mechanisms for testing and debugging grammars, including a comprehensive graphical user interface. Additional non-interactive command-line tools can be used for automated grammar processing. Moreover, the XDK contains a solver for XDG, the extensible principle library, and an interface to external knowledge sources to (e.g. statistically) guide the search for solutions.

4.1 Concrete Syntax

The XDK defines three concrete syntaxes for grammar specification, each of which fulfills a different purpose. The User Language (UL) is an input language for manual grammar development. The XML language (XML) is based on XML, and is particularly well suited for automated grammar development (e.g. automatic grammar induction from corpora). The Intermediate Language (IL) is a record-based language tailored for Mozart/Oz and for further processing within the XDK, but is neither readable (as the UL), nor suited for automated processing outside Mozart/Oz (as the XML). The XDK offers functionality to convert the different languages into each other, e.g. to make XML grammars readable by converting them into the UL.

We illustrate how XDG grammars look like by a miniature example grammar using UL syntax. XDG grammars are split up into two main parts: 1) the header, and 2) the lexicon. The header includes type definitions (e.g. the set of edge labels or the type of a lexical entry), and specifies the principles used from the principle library. The lexicon is a metagrammatical lexicon specification. We display the header of the example grammar in Fig. 2, and the lexicon in Fig. 3.

The `usedim` keyword activates dimensions. In the example, it activates the `syn`, `sem` and `lex`[4]. In the `defdim` sections, we define the types pertaining to the respective dimensions of the grammar. `deftype` defines a type and binds it to a name, e.g. `syn.label` to {`det subj obj vcomp`}. These names can be dereferenced by their name. `defentrytype` defines the type of a lexical entry, and `deflabeltype` the type of edge labels. The `useprinciple` keyword indicates the use of a principle and `dims` binds dimension variables to actual dimensions. E.g. the linking principle `principle.linking` binds dimension variable `D1` to `sem`, and `D2` to `syn`. The `output` and `useoutput` keywords specify the output functors to visualize analyses.

The UL syntax of the lexicon specification is close to the abstract syntax presented before. `defclass` defines lexical classes (clauses). E.g. `defclass "det" Word` defines the lexical class named `det`, and with one argument Word. Lexical classes can be dereferenced by giving their name and the required arguments. E.g. `"det" {Word: "every"}` dereferences class `det` and binds its argument `Word` to `every`. `defentry` defines a set of lexical entries. Disjunction ($\vee$) is written `|`.

[4] The `lex` dimension is not a real XDG dimension—it is used solely identify a word with each lexical entry.

```
usedim syn
usedim sem
usedim lex
%%
defdim syn {
  deftype "syn.label" {det subj obj vcomp}
  deftype "syn.entry" {in: valency("syn.label")
                       out: valency("syn.label")}
  defentrytype "syn.entry"
  deflabeltype "syn.label"
%%
  useprinciple "principle.graph" { dims {D: syn} }
  useprinciple "principle.tree" { dims {D: syn} }
  useprinciple "principle.valency" { dims {D: syn} }
%%
  output "output.dag"
  useoutput "output.dag"
}
defdim sem {
  deftype "sem.label" {det ag pat prop}
  deftype "sem.entry" {in: valency("sem.label")
                       out: valency("sem.label")
                       link: map("sem.label" iset("syn.label"))}
  defentrytype "sem.entry"
  deflabeltype "sem.label"
%%
  useprinciple "principle.graph" { dims {D: sem} }
  useprinciple "principle.dag" { dims {D: sem} }
  useprinciple "principle.valency" { dims {D: sem} }
  useprinciple "principle.linking" { dims {D1: sem
                                           D2: syn} }
%%
  output "output.dag"
  useoutput "output.dag"
}
%%
defdim lex { defentrytype {word: string} }
```

Fig. 2. The header of the example grammar

4.2 Error Detection

The XDK offers various ways to detect errors, including a very fast static grammar type checker. This type checker is implemented for the IL, and hence also for the UL and the XML languages (since they are always compiled into the IL). The type checker also detects cycles in the definition of lexical classes.

4.3 Graphical Interfaces

The XDK comprises a comprehensive graphical user interface (GUI) for convenient access to all the functionality of the system. The GUI is most useful for debugging grammars, e.g. by switching off any of the principles to find out which constraints have ruled out desired analyses. The GUI visualizes the solver search tree using the *Oz Explorer* or optionally Guido Tack's new Explorer replacement *IOzSeF*, and can visualize partial and total analyses using functors from an extensible *output library* of output functors, including a graphical DAG display, a

```
defclass "n" {
  dim syn {in: {subj?} | in: {obj?}}
  dim sem {in: {ag*} | in: {pat*}}}

defclass "cn" Word {
  "n"
  dim syn {out: {det!}}
  dim sem {out: {det!}}
  dim lex {word: Word}}

defclass "pn" Word {
  "n"
  dim lex {word: Word}}

defclass "modal" Word {
  dim syn {in: {}
           out: {subj! vcomp!}}
  dim sem {in: {}
           out: {ag! prop!}
           link: {ag: {subj}
                  prop: {vcomp}}}
  dim lex {word: Word}}
```

```
defclass "det" Word {
  dim syn {in: {det?}}
  dim sem {in: {det?}}
  dim lex {word: Word}}

defclass "trans" Word {
  dim syn {in: {vcomp?}
           out: {obj!}}
  dim sem {in: {prop?}
           out: {ag! pat!}
           link: {ag: {subj}
                  pat: {obj}}}
  dim lex {word: Word}}
%%
defentry { "det" {Word: "every"} }
defentry { "cn" {Word: "programmer"} }
defentry { "modal" {Word: "should"} }
defentry { "trans" {Word: "like"} }
defentry { "pn" {Word: "Mozart"} }
```

Fig. 3. The lexicon of the example grammar

detailed display of the underlying analysis using the *Oz Inspector*, LaTeX output (as used to create Fig. 1), or an XML-based output for further processing. We depict the main GUI window and the Oz Explorer in Fig. 4, and an example XDG analysis as displayed by the DAG output functor in Fig. 5.

4.4 Solver

The XDK solver makes use of Denys Duchier's axiomatization of dependency parsing [14, 17], and turns it into a completely modular, extensible *principle library*. Principles are composed from sets of constraint functors: For instance the valency principle is composed from the *in constraint* and the *out constraint*, constraining resp. the incoming and outgoing edges of each node. The starting sequence of the constraints can be regulated by global *constraint priorities*. This can help gaining efficiency. New principles and new constraints can easily be added and integrated into the XDK, which makes it an ideal launchpad for new linguistic theories.

4.5 Preferences and Search

Following ideas by Thorsten Brants and Denys Duchier, Dienes et al. [19] introduce the idea to guide the search for solutions of the XDK solver by external knowledge sources called *Oracles*. Oracles interact with the XDK solver by sockets, and are based either on statistical information or heuristics. The XDK supports the use of Oracles using a standard architecture for Oracles developed by Marco Kuhlmann and others.

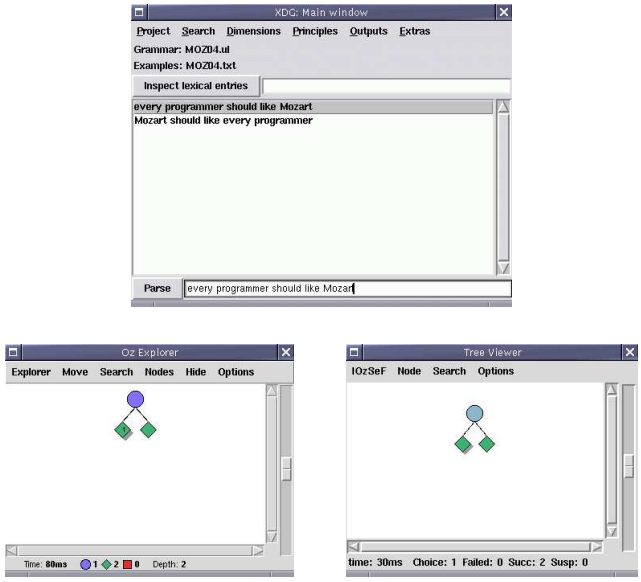

Fig. 4. The main window of the GUI (top), the Oz Explorer (bottom left), and IOzSeF (bottom right)

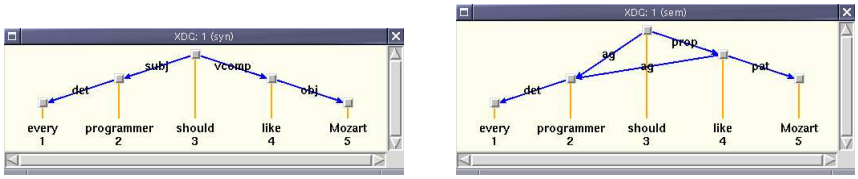

Fig. 5. The XDG analysis displayed by the DAG output functor

5 Mozart Implementation

In this section, we discuss selected aspects of our implementation of the XDK in Mozart/Oz.

5.1 Constraint Programming

Constraint programming is used to enumerate graph models of graph descriptions. The techniques used for XDG rely on ideas from TDG[17]. We illustrate them here in order to illustrate XDG's requirements on constraint programming.

Finite Set Constraints are used to model graph configuration problems. For example, the daughters of node w that can be reached by traversing an edge labeled `obj` are represented by the set variable $\texttt{obj}(w)$. A valency specification `obj`? can be enforced by posting the cardinality constraint $|\texttt{obj}(w)| \in \{0, 1\}$

Selection Constraints are used to efficiently handle ambiguity. Typically, a word w has multiple lexical entries $L_1, \ldots, L_n$. If we introduce a variable E_w to denote the lexical entry that is ultimately selected among them, and an integer variable I_w to denote it's position in that sequence, then we can relate these quantities by a selection constraint:

$$E_w = \langle L_1, \ldots, L_n \rangle [I_w]$$

with the declarative semantics that $E_w = L_{I_w}$. The basic selection constraints implemented for finite domains and finite sets can trivially be lifted to record types.

Deep Guards in Disjunctive Propagators. The construct `or` G_1 `[]` G_2 `end` is used to enforce complex mutually exclusive well-formedness conditions. For example that either (G_1) a certain tree edge exists and it satisfies some additional condition, or it does not exist (G_2). For every possible edge, there is a disjunctive propagator to monitor these alternatives concurrently.

5.2 Programming Environment

The XDK makes use of modules from the *MOzart Global User Library* (MOGUL), and applies *ozmake* for convenient compilation and deployment (again into MOGUL). The principle and output libraries are realized using dynamically linked *functors*.

The grammar compiler utilizes two parsers: 1) a flexible LR/LALR parser generator (fully written in Mozart by Denys Duchier) for parsing the UL, and 2) the fast XML parser by Denys Duchier from the Mozart Standard Library for parsing grammars written in XML. Per default, grammars are stored as pickles, but the XDK can also make use of Denys Duchier's interface to the *GNU GDBM database library*, with which very large grammars can be handled more efficiently.

The graphical user interface of the XDK is written using the Tcl/Tk interface of Mozart/Oz. Moreover, the XDK utilizes the Oz Explorer and optionally *IOzSeF* (by Guido Tack) to visualize the solver search tree, and the Oz Inspector to display XDG structures in more detail.

6 Conclusion

We have presented the XDG Development Kit (XDK), and described its lexicon specification language. The XDK includes a large number of grammar development tools fully implemented in Mozart/Oz, making use of its flexible multi-paradigmatic nature. No other programming system provides the required expressiveness to combine set constraints, selection constraints, and deep guards as used in the solver of the XDK. Furthermore, it was very easy to add a GUI, and support for multiple input languages, including an XML-based one, and much more.

A stable and thoroughly tested version of the XDK is freely available in the MOGUL library, including a comprehensive manual covering the entire system in quite some detail (more than 200 pages).

References

1. Bresnan, J., Kaplan, R.: Lexical-Functional Grammar: A Formal System for Grammatical Representation. In Bresnan, J., ed.: The Mental Representation of Grammatical Relations. The MIT Press, Cambridge/USA (1982) 173–281
2. Kay, M.: Functional Grammar. In C. Chiarello et al., ed.: Proceedings of the 5^{th} Annual Meeting of the Berkeley Linguistics Society. (1979) 142–158
3. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago/USA (1994)
4. Lambek, J.: The Mathematics of Sentence Structure. American Mathematical Monthly (1958) 154–170
5. Steedman, M.: The Syntactic Process. MIT Press (2000)
6. Joshi, A.K., Levy, L., Takahashi, M.: Tree Adjunct Grammars. Journal of Computer and System Sciences **10** (1975)
7. Joshi, A.K.: How much context-sensitivity is necessary for characterizing structural descriptions—Tree Adjoining Grammars. In Dowty, D., Karttunen, L., Zwicky, A., eds.: Natural Language Processing—Theoretical, Computational and Psychological Perspectives. Cambridge University Press, New York/USA (1985)
8. Tesnière, L.: Eléments de Syntaxe Structurale. Klincksiek, Paris/FRA (1959)
9. Mel'čuk, I.: Dependency Syntax: Theory and Practice. State Univ. Press of New York, Albany/USA (1988)
10. Copestake, A.: Implementing Typed Feature Structure Grammars. CSLI Publications (2002)
11. XTAG Research Group: A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania (2001)
12. Kaplan, R.M., Maxwell, J.T.: LFG Grammar Writer's Workbench. Technical report, Xerox PARC (1996)
13. Smolka, G., Uszkoreit, H.: NEGRA Project of the Collaborative Research Centre (SFB) 378 (1996–2001) Saarland University/GER.
14. Duchier, D.: Axiomatizing Dependency Parsing Using Set Constraints. In: Proceedings of MOL6, Orlando/USA (1999)
15. Duchier, D., Debusmann, R.: Topological Dependency Trees: A Constraint-Based Account of Linear Precedence. In: Proceedings of ACL 2001, Toulouse/FRA (2001)
16. Debusmann, R., Duchier, D., Koller, A., Kuhlmann, M., Smolka, G., Thater, S.: A Relational Syntax-Semantics Interface Based on Dependency Grammar. In: Proceedings of COLING 2004, Geneva/SUI (2004)
17. Duchier, D.: Configuration of Labeled Trees under Lexicalized Constraints and Principles. Research on Language and Computation **1** (2003) 307–336
18. Duchier, D.: MOGUL: the MOzart Global User Library (2004) http://www.mozart-oz.org/mogul/.
19. Dienes, P., Koller, A., Kuhlmann, M.: Statistical A* Dependency Parsing. In: Prospects and Advances in the Syntax/Semantics Interface, Nancy/FRA (2003)