Enhanced Multimodal Grammar Library

Peter Ljunglöf Gabriel Amores Håkan Burden
Pilar Manchón Guillermo Pérez Aarne Ranta

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 1.5

15/08/06

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	Jun 06
Actual date of delivery	15/08/06
Deliverable number	1.5
Deliverable title	Enhanced Multimodal Grammar Library
Type	Report
Status & version	Public Final
Number of pages	55 (excluding front matter)
Contributing WP	1
WP/Task responsible	UGOT
Other contributors	USE
Author(s)	Peter Ljunglöf, Gabriel Amores, Håkan Burden, Pilar Manchón, Guillermo Pérez and Aarne Ranta
EC Project Officer	Evangelia Markidou
Keywords	grammar, multilingual, multimodal, multimodal fusion, OWL, ontology, dialogue systems, Grammatical Framework, TrindiKit, GoDiS, DelfosNCL

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, http://www.talk-project.org

©2006, The Individual Authors.

No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.

Contents

Summary 1

1 Introduction **2**

1.1 OWL, the Web Ontology Language 4

1.2 An overview of the Grammatical Framework 5

1.2.1 Main features of GF 5

1.3 The GF resource grammar library 6

1.3.1 Implemented languages and linguistic coverage 7

1.3.2 A small example 8

1.3.3 Inflection paradigms 9

1.3.4 Syntax rules 9

1.3.5 Syntactic structures on the sentence-level 10

1.3.6 Sub-sentential syntactic structures 11

1.3.7 Multimodality 13

2 Generating Multilingual Grammars from OWL Ontologies in MIMUS **14**

2.1 Introduction 14

2.2 Automatic grammar generation 15

2.2.1 Generating grammars from ontologies 16

2.3 Solution overview 16

2.4 Configuration files 17

2.5 Overview of the algorithm 17

2.6 Showcases 19

2.6.1 Sample rules 19

2.6.2 Capturing multimodality 21

2.6.3 Capturing multilinguality 22

2.7 Conclusions and future work 24

3 Ontologies and Grammatical Framework **25**

3.1 Relating existing ontologies with abstract GF grammars 25

3.1.1 Classes and individuals 25

3.1.2 Properties 27

3.1.3 General OWL axioms and restrictions . . . 29
3.1.4 Ontologies and concrete syntax . . . 29
3.2 Incorporating the Mimus in-home ontology . . . 30
3.2.1 The Mimus ontology as an abstract GF grammar . . . 30
3.2.2 Mimus configuration files as GF concrete syntax . . . 31
3.2.3 Discussion . . . 32
3.3 GF abstract syntax as OWL ontologies . . . 33
3.3.1 Context-free grammars as OWL ontologies . . . 33
3.3.2 GF abstract syntax in OWL . . . 33
3.3.3 Optimizing the grammar ontology . . . 34
3.3.4 The GF resource grammar as an OWL ontology . . . 35
3.3.5 Ontology editing as a multilingual authoring system . . . 36
3.4 Using ontologies to specify GoDiS dialogue domains . . . 36
3.4.1 The GF resource grammar . . . 36
3.4.2 Actions and predicates . . . 36
3.4.3 Sorts and indivuduals . . . 37

4 The Enhanced Multimodal Grammar Library **39**
4.1 Separating system and user utterances . . . 39
4.2 The domain-independent grammars . . . 40
4.3 The domain-specific grammars for system utterances . . . 41
4.4 The domain-specific grammars for user utterances . . . 44
4.5 Integrating multimodality in the grammars . . . 46
4.6 Extending the GF grammar library . . . 47

5 Summary and Conclusions **49**

Bibliography **51**

A The enhanced multimodal grammar library **53**
A.1 Downloading the grammar library . . . 53
A.2 Installation instructions . . . 53
A.3 Testing the grammars . . . 54

Summary

The ISU approach uses abstract representations for dialogue states and update rules which allow the generic characterisation of flexible dialogue strategies. This enables the same code for dialogue management techniques to be used for different natural languages and for different domains.

In this deliverable we discuss how dialogue systems, and especially grammars for dialogue systems, can be related to existing knowledge representation systems. We focus on one specific language for knowledge representation, the Web Ontology Language (OWL), which is a W3C standard for ontology descriptions for knowledge representation. Since it is a standard there is already much work done on relating OWL to other ontology formalisms.

We show how to generate the domain-specific utterances for a device-oriented dialogue system from an ontology describing the devices. An example system using this technique is Mimus, a dialogue system with which one can control different devices in a home, such as lights, alarms and washing machines. All information about devices is specified in an ontology, which can be updated on the fly with e.g. new devices or new information about existing devices.

We also show that all domain-specific utterances in a domain for the GoDiS dialogue system can be specified as a single ontology. By representing the utterances as abstract syntax trees in a Grammatical Framework (GF) resource grammar, the representation of utterances is language-independent. The resource grammar we are using has a coverage comparable to the Core Language Engine, and exists for 11 different languages, including the TALK languages English, German, Spanish, and Swedish, but also the non-Indoeuropean languages Finnish, Russian and Arabic. Since the grammar is multilingual, this means that the only thing that has to be done to localize a dialogue application to a new language is to write a lexicon for the domain-specific entities.

In this deliverable we also describe the final status of the full TALK Grammar Library, which is written in GF. The structure is modified so that the domain-specific parts of a grammar can be described in an OWL ontology, and to further simplify localization to new languages, domains and modalities.

Chapter 1

Introduction

This deliverable concerns the development of technology for making use of ontological knowledge in dialogue system grammars. We have focussed on one specific language for knowledge representation, the Web Ontology Language (OWL), which is a W3C standard for ontology descriptions for knowledge representation. We discuss how the abstract syntax of the Grammatical Framework (GF) can be used as an ontology specification language.

The main concern of the deliverable is to exploit techniques for automatically generating multilingual and multimodal grammars from an ontology. We present two similar ideas which are implemented in the dialogue systems Mimus and GoDiS. In Mimus the dialogue system grammars are automatically generated from an ontology of devices, by supplying a configuration file explaining the linguistic details. In GoDiS the whole dialogue domain can be specified as an ontology, from which GF grammars are generated. The rest of the dialogue application can also be generated from the ontology, which is discussed in TALK deliverable D2.2 (Milward et al., 2006).

Finally we describe the final structure of the Enhanced Multimodal Grammar Library. In making the grammar library we exploit the advantages of the ISU approach. The ISU approach utilizes structured Information States to keep track of dialogue context information. These Information States can be read and updated by several different modules which access precisely the information that they need. This enables a modular architecture which allows generic solutions for dialogue technology. For example,

- different language modules can interact with essentially similar Information States, enabling rapid porting of dialogue systems from one language to another and the creation of multilingual dialogue systems;
- coding of dialogue behaviour is supported independently of language and domain, thus allowing for the rapid porting of dialogue systems to different domains;
- the use of structured Information States allows straightforward implementation of flexible dialogue systems which can access and modify information in the Information State in different sequences and by varying means.

In this deliverable, as well as in the earlier TALK deliverables D1.1, D1.2a and D1.2b (Ljunglöf et al., 2005; Bringert et al., 2005; Ljunglöf et al., 2006), we show that by using an abstract representation for grammars, we can further enable rapid porting of dialogue systems between languages, domains and

modalities. The main tool in defining such grammars is the Grammatical Framework (GF), which is used in collaboration by UGOT, UEDIN and UCAM for making ISU-based dialogue systems.

Comparison with the current state of the art

Milward and Beveridge (2003) notes that general use of ontologies within dialogue systems is relatively rare, and discuss in what different ways ontological domain knowledge can be used by linguistic components in a dialogue system. Although they discuss generation of simple recognition grammars using synonyms from the ontology and extend this to recognition grammars for noun phrases, the approach does not extend to the generation of language models for complex commands or questions.

Some recent work has focussed on incorporating ontological knowledge into the parsing process. Coppi et al. (2005) use a two-level grammar which refers to concepts in the ontology, to build a parser translating natural language input to formulae in description logics. Bernstein et al. (2006) use a similar approach, with a static grammar for recognizing domain-independent linguistic structures and a dynamic grammar which is automatically generated from an OWL ontology. However, these systems are simple question-answering systems and not general dialogue systems.

In particular, there has been no previous work on incorporating ontological knowledge into recognition and generation grammars for ISU-based dialogue systems. The main result of this deliverable is that we present different solutions of how this integration of ontologies and grammars into ISU-based dialogue systems can be made. The solutions are also practical since they are implemented in existing dialogue systems.

One of the main advantages with integrating ontologies and ISU-based dialogue systems is that it becomes simple and fast to implement new languages and/or develop new dialogue domains. In fact, a complete dialogue application for the GoDiS dialogue system can be specified as a single OWL ontology. From this ontology all necessary dialogue system files can be generated, which is shown in TALK deliverable D2.2 (Milward et al., 2006), as well as the necessary recognition and generation grammars, which is shown in this deliverable.

This is similar to Denecke (2002) which describes a framework for rapid prototying of form-filling dialogue systems. Since we are using an ISU-based approach, we in addition get all the benefits described above, such as language-independence and dialogue flexibility. Furthermore, by converting the ontologies to Grammatical Framework grammars, we can create a multilingual dialogue system by only writing a domain-dependent lexicon for each language.

Layout of the deliverable

We begin by giving a general description of OWL, the Web Ontology Language, followed by a summary of the Grammatical Framework. We then describe the multilingual GF resource grammar, which has a wide linguistic coverage of 11 languages, and some more in the making.

In chapter 2 we describe how multilingual grammars for the Mimus dialogue system are generated from an OWL ontology describing the devices in the domain. The generation is controlled by a configuration file where the linguistic details are given.

In chapter 3 we discuss how OWL is related to GF abstract syntax. We show how an OWL ontology can be translated to a GF abstract grammar, and how multilingual presentations of the concepts in the ontology can be implemented as concrete syntaxes. We also discuss how a GF grammar can be converted

to an OWL ontology, by which an ontology editor can be used as a multilingual authoring tool. The final section describes how a GoDiS dialogue domain can be implemented as an OWL ontology, from which all necessary GoDiS files and GF grammars can be generated.

Finally, chapter 4 consists of a description of the final structure of the Enhanced Multimodal Grammar Library, which is implemented in GF as a front-end to the generic dialogue system GoDiS built within TrindiKit. The library also includes three example dialogue domains – a calendar application called AgendaTalk, an MP3 player application called DJGoDiS, and a Tram information application called TramGoDiS. The library is designed for making is easy to add new dialogue domains, source languages, and input and output modalities.

1.1 OWL, the Web Ontology Language

OWL, the Web Ontology Language, is a W3C standard for describing ontologies.[1] We are not using the full flavour of OWL, only the main components are needed. The main reason why we have chosen OWL instead of any other ontology description language is that it is a standard.

OWL has three main components – classes, individuals and properties:

Classes

Classes correspond to sets of individuals. The main relation between classes is entailment – that classes can be subclasses of other classes. There are no constraints on the subclass relation, which means that a class can e.g. be a subclass of several classes.

It is possible to form combined classes, most notably the intersection or the union of several classes. Classes can also be declared to be equivalent, or disjoint.

Individuals

Individuals are class elements. Just as in set theory, an individual can be an element of several classes. This is equivalent to the element being in the intersection class.

Properties

Properties correspond to relations between individuals. There are three kinds of properties – object, datatype and annotation properties. An object property has a domain and a range, which are themselves classes. The range of a datatype property is not a class, but instead a datatype, e.g. a number or a string. Annotation properties correspond to extra-logical properties, e.g. comments or version information.

Instead of writing an instance of a property as $P(a,b)$ we often say that "a has the property $P(b)$". I.e. properties are seen as directed, from the domain to the range. Properties can also be declared to be functional, transitive, symmetric, and more. A property can also be a subproperty of another.

There are more than classes, individuals and properties in OWL. It is e.g. possible to create restrictions on classes and properties be logical formulae. There are three different levels of OWL – OWL Lite, OWL

[1] http://www.w3.org/2004/OWL/

DL[2] and OWL Full – with different expressive power, and theorem proving complexity. The reader is referred to the OWL Language Reference (Dean and Schreiber, 2004) and OWL Semantics and Abstract Syntax (Patel-Schneider et al., 2004) for more information.

Our ontologies are very simple, and fit into OWL Lite, except for one thing: Some of our properties have the class of classes as their range, meaning that in these cases ordinary classes act as individuals. This is only allowed in OWL Full, which is no serious problem we do not do any formal reasoning on our ontologies.

1.2 An overview of the Grammatical Framework

Grammatical Framework (GF) has been extensively described in TALK deliverables D1.1, D1.2a and D1.2b (Ljunglöf et al., 2005; Bringert et al., 2005; Ljunglöf et al., 2006), so in this section we only give a short overview of the main features of the formalism. In section 1.3 we describe the structure of the multilingual resource grammar.

1.2.1 Main features of GF

The main idea with the Grammatical Framework is that it maintains a clean separation of abstract and concrete syntax.

Abstract syntax

The abstract syntax is a context-free grammar[3] without terminals, and where each rule has a unique name. An abstract rule in GF is written as a typed function. For a person used to phrase structure grammars, this syntax might look awkward, and another equivalent notation is the one used in Multiple Context-Free Grammars (MCFG) (Seki et al., 1991):

$$\begin{aligned} f : A_1 \to \cdots \to A_n \to A & \qquad \text{(GF notation)} \\ A \to f[A_1, \ldots, A_n] & \qquad \text{(MCFG notation)} \end{aligned}$$

The categories and rules are specified in GF by `cat` and `fun` declarations:

```
cat A; A1; ...; An;
fun f : A1 -> ... -> An -> A;
```

The abstract grammar defines a language of trees over names called *terms*, where $\mathcal{L}(A)$ are all terms with category A:

$$\begin{aligned} \mathcal{L}(A) \quad = \quad \{ f(t_1, \ldots, t_n) \mid & f : A_1 \to \cdots \to A_n \to A, \\ & t_1 \in \mathcal{L}(A_1), \ldots, t_n \in \mathcal{L}(A_n) \} \end{aligned}$$

[2] The term *DL* stands for *Description Logic* (Baader et al., 2003)

[3] GF also supports non-context-free rules, by the use of dependent types and higher-order functions. Furthermore, it is possible to define reduction rules via the `def` declaration. See Ranta (2004) for further details.

Concrete syntax

One abstract grammar can have several corresponding concrete grammars, which makes GF a natural multilingual grammar formalism. The concrete grammar specifies how the abstract grammar rules should be linearized in a compositional manner. Each abstract category A has a corresponding *linearization type* A°, and each linearization rule $f : A_1 \to \cdots \to A_n \to A$ has a corresponding *linearization function* f° over linearization types:

$$f^\circ \quad : \quad A_1^\circ \to \cdots \to A_n^\circ \to A^\circ$$

The linearization types and linearization functions are specified in GF by `lincat` and `lin` declarations:

```
lincat A = A°;
lin f x1  xn = f°(x1,...,xn);
```

The linearization of a tree $f(t_1,\ldots,t_n) \in \mathcal{L}(A)$ is:

$$[\![f(t_1,\ldots,t_n)]\!] \quad = \quad f^\circ([\![t_1]\!],\ldots,[\![t_n]\!])$$

The expressive power of GF comes from the rich type system for specifying linearization types. It supports discontinuous constituents, inflection tables and inflectional parameters, in addition to ordinary string sequences.

Resource syntax and modules

A final feature of GF is the possibility of definining *parameters* and *operations*, which is done in a resource module. The parameters can be used for inflection tables, and operations can be used as macros for simplifying grammar writing.

GF has a rich module system which supports extension, importing and instantiation of other modules. A concrete grammar can be used as a resource module in other grammars, which gives a notion of grammar composition. This makes it possible to use a wide coverage grammar such as the GF resource grammar described in section 1.3, when writing a specialized domain-specific grammar. The GF compiler takes care of extracting only those parts of the resource grammar that are interesting for the domain, which will give us efficient parsing of the specialized grammar.

1.3 The GF resource grammar library[4]

Even though it is easy to write simple GF grammars and make them run in a couple of languages, scaling up to bigger language fragments can require considerable work. The main part of this work is linguistic: the grammar writer has to know a lot about the morphology and the syntax of the target languages to get the concrete syntax correct. The solution provided by GF to this problem is that of library-based software engineering: the work can be divided between resource grammarians working on linguistic details,

[4] This section is a rewritten and compressed version of the user's manual of the resource grammar library: http://www.cs.chalmers.se/~aarne/GF/doc/resource.pdf

and application grammarians working on domain semantics and lexicon. GF's module system helps to maintain this division of labour.

The GF Resource Grammar Library has been developed during the last five years to serve as a standard library of GF and provide the linguistic details for application grammars on different domains. The development has been guided by two major applications using and testing the resource grammars:

- KeY,[5] an authoring system for software specifications in the formal language OCL, as well as in English and German
- WebALT,[6] a system for generating mathematical exercises from MathML representations (currently English, Finnish, French, Italian, Spanish, and Swedish)

In the TALK project, the resource grammar library has been used for parsing and generation in dialogue systems, currently in English, Spanish, and Swedish Johansson (2006).

In this chapter, we give a brief introduction to the GF Resource Grammar Library, focusing on how it is used when writing application grammars. We presuppose knowledge of GF and its module system, knowledge that can be acquired e.g. from the GF tutorial.[7] For the details of the GF Resource Grammar, we refer to its own documentation:

- printable User's Manual (from which this chapter is adapted):
  `http://www.cs.chalmers.se/~aarne/GF/doc/resource.pdf`
- on-line documentation:
  `http://www.cs.chalmers.se/~aarne/GF/lib/resource-1.0/doc/`

1.3.1 Implemented languages and linguistic coverage

The GF Resource Grammar Library contains grammar rules for 11 languages, plus some more under construction. These languages are `Ara`bic,[8] `Dan`ish, `Eng`lish, `Fin`nish, `Fre`nch, `Ger`man, `Ita`lian, `Nor`wegian, `Rus`sian, `Spa`nish, and `Swe`dish. For each language, the library provides

- a complete set of inflectional paradigms
- a comprehensive set of syntax rules, with structures such as
  - texts, punctuation
  - declaratives, questions, imperatives, one-phrase utterances
  - predication in different tenses and moods
  - verb phrases constructed from verbs and adjectives with different subcategorization patterns
  - noun phrases formed by determiners and from common nouns, proper names, adjectives, numerals, and pronouns

[5] `http://www.key-project.org/`
[6] `http://webalt.math.helsinki.fi/`
[7] `http://www.cs.chalmers.se/~aarne/GF/doc/tutorial/gf-tutorial2.html`
[8] The Arabic grammar does not cover the full resource API yet.

- relative clauses and wh-questions
- coordination on different levels

As a benchmark for coverage, we have used the CLE (Core Language Engine) grammars as described in Rayner et al. (2000).

1.3.2 A small example

To give an example of how the library is applied, consider music playing devices. In the application, we may have a semantical category `Kind`, examples of `Kinds` being `Song` and `Artist`. In German, for instance, `Song` is linearized into the noun *"Lied"*, but knowing this is not enough to make the application work, because the noun must be produced in both singular and plural, and in four different cases. By using the resource grammar library, it is enough to write,

```
lin Song = reg2N "Lied" "Lieder" neuter;
```

and the eight forms are correctly generated. The resource grammar library contains a complete set of inflectional paradigms (such as `reg2N` here), enabling the definition of any lexical items.

The resource grammar library is not only about inflectional paradigms - it also has syntax rules. The music player application might also want to modify songs with properties, such as *"American"*, *"old"*, *"good"*. The German grammar for adjectival modifications is particularly complex, because adjectives have to agree in gender, number, and case, and also depend on what determiner is used (*"ein amerikanisches Lied"* vs. *"das amerikanische Lied"*). All this variation is taken care of by the resource grammar function `AdjCN`,

```
fun AdjCN : AP -> CN -> CN;
```

The resource library API is devided into language-specific and language-independent parts. To put it roughly,

- the lexicon API is language-specific
- the syntax API is language-independent

Thus, to render the above example in French instead of German, we need to pick a different linearization of `Song`,

```
lin Song = regGenN "chanson" feminine;
```

But to linearize the rule that modifies a kind with a property, we can use the very same rule in German and French. The resource function `AdjCN` has different implementations in the two languages (e.g. different word orders), but the application programmer need not worry about the difference.

1.3.3 Inflection paradigms

Inflection paradigms are defined separately for each language *Lng* in the module `Paradigms`*Lng*. For the sake of convenience, every language implements these five paradigms:

```
oper regN  : Str -> N;     -- regular nouns
     regA  : Str -> A;     -- regular adjectives
     regV  : Str -> V;     -- regular verbs
     regPN : Str -> PN;    -- regular proper names
     dirV  : V   -> V2;    -- direct transitive verbs
```

It is often possible to initialize a lexicon by just using these functions, and later revise it by using the more involved paradigms. For instance, in German we cannot use `regN "Lied"` for `Song`, because the result would be a Masculine noun with the plural form *"Liede"*. The individual `Paradigms` modules tell what cases are covered by the regular heuristics.

As a limiting case, one could even initialize the lexicon for a new language by copying the English (or some other already existing) lexicon. This would produce language with correct grammar but with content words directly borrowed from English – maybe not so strange in certain technical domains.

1.3.4 Syntax rules

Syntax rules should be looked for in the abstract modules defining the API. There are around 10 such modules, each defining constructors for a group of one or more related categories. For instance, the module `Noun` defines how to construct common nouns, noun phrases, and determiners. Thus the proper place to find out how nouns are modified with adjectives is `Noun`, because the result of the construction is again a common noun.

Browsing the libraries is helped by the gfdoc-generated HTML pages. However, this is still not easy, and the most efficient way is probably to use the parser. To find out which resource function implements a particular structure, one can just parse a string that exemplifies this structure. For instance, to find out how sentences are built using transitive verbs, write

```
> i english/LangEng.gf

> p -cat=Cl "she loves him"
PredVP (UsePron she_Pron) (ComplV2 love_V2 (UsePron he_Pron))
```

Parsing with the resource grammar has an acceptable speed, for English and the Scandinavian languages. In the current implementation, parsing for other languages can be inefficient. However, examples parsed in one language can always be linearized into other languages:

```
> i italian/LangIta.gf
> l PredVP (UsePron she_Pron) (ComplV2 love_V2 (UsePron he_Pron))
lo ama
> p -cat=Cl -lang=LangEng "she loves him" | l -lang=LangIta
lo ama
```

1.3.5 Syntactic structures on the sentence-level

Texts, phrases and utterances

The outermost linguistic structure is `Text`. A `Text` is composed from a sequence of Phrases (`Phr`) followed by punctuation marks. Phrases are built from Utterances (`Utt`), which in turn are declarative sentences, questions, or imperatives – but there are also "single-phrase utterances" consisting of noun phrases or other subsentential phrases. The difference between Phrase and Utterance is mostly technical: a Phrase is an Utterance with an optional leading conjunction (*"but"*) and an optional tailing vocative (e.g. *"John"* or *"please"*).

Sentences and clauses

The richest of the categories below Utterance is `S`, Sentence. A Sentence is formed from a Clause (`Cl`), by fixing its Tense, Anteriority, and Polarity. The difference between Sentence and Clause is thus also rather technical. For example, each of the following strings has a distinct syntax tree in the category Sentence,

John walks
John doesn't walk
John walked
John didn't walk
John has walked
John hasn't walked
John will walk
John won't walk
...

whereas in the category Clause all of them are just different forms of the same syntax tree.

In figure 1.1 there are some examples of the results of replacing parts of the syntax tree for the sentence *"John walks"*.

Questions

The categories Sentence (S) and Clause (Cl) have question counterparts in the categories QS and QCl. A Question Sentence is formed from a Question Clause by fixing its Tense, Anteriority and Polarity, in the same way as a Sentence is formed from a Clause.

Question clauses can be formed from clauses (yes/no-questions) or by using an interrogative (wh-questions). The interrogatives are pronouns (`IP`, *"who"*, *"which song"*), adverbials (`IAdv`, *"why"*) and complements (`IComp`, *"where"*).

Other sentence-level categories

Apart from sentences, clauses and questions, the resource grammar library contains several other sentence-level categories:

- Relative clauses (`RCl`) and relative sentences (`RS`), *"who loves John"*

Original syntax tree for the sentence "John walks":
`UseCl TPres ASimul PPos (PredVP (UsePN john_N) (UseV walk_V))`

Subtree		Replacement	Resulting sentence
`TPres`	⇒	`TPast`	*"John walked"*
`TPres`	⇒	`TFut`	*"John will walk"*
`ASimul`	⇒	`AAnter`	*"John has walked"*
`PPos`	⇒	`PNeg`	*"John doesn't walk"*
`john_PN`	⇒	`mary_PN`	*"Mary walks"*
`UsePN john_PN`	⇒	`UsePron it_Pron`	*"it walks"*
`walk_V`	⇒	`sleep_V`	*"John sleeps"*
`UseV walk_V`	⇒	`ComplV2 love_V2 somebody_NP`	*"John loves somebody"*

Figure 1.1: Results of replacing parts of the syntax tree for the Sentence "John walks"

- Slash clauses (`Slash`), *"(whom) she sees"*
- Imperatives (`Imp`), *"watch this"*
- Embedded sentences (`SC`), *"whether you go"*

1.3.6 Sub-sentential syntactic structures

The linguistic phenomena mostly discussed in both traditional grammars and modern syntax belong to the level of Clauses. At this level, the major categories are `NP` (noun phrase) and `VP` (verb phrase). A Clause typically consists of just an `NP` and a `VP`. The internal structure of both `NP` and `VP` can be very complex, and these categories are mutually recursive: not only can a `VP` contain an `NP` – [`VP` *loves* [`NP` *somebody*]] – but also an `NP` can contain a `VP` – [`NP` *every man* [`RS` *who* [`VP` *walks*]]].
Most of the resource modules thus define functions that are used inside `NP`s and `VP`s. Here is a brief overview:

The `Noun` module

How to construct `NP`s. The main three mechanisms for constructing `NP`s are

- from proper names: *"John"*
- from pronouns: *"we"*
- from common nouns by determiners: *"this man"*

The `Noun` module also defines the construction of common nouns. The most frequent ways are

- lexical noun items: *"man"*

- adjectival modification: *"old man"*
- relative clause modification: *"man who sleeps"*
- application of relational nouns: *"successor of the number"*

The `Verb` module

How to construct `VP`s. The main mechanism is verbs with their arguments, for instance,

- one-place verbs: *"walks"*
- two-place verbs: *"loves Mary"*
- three-place verbs: *"gives her a kiss"*
- sentence-complement verbs: *"says that it is cold"*
- VP-complement verbs: *"wants to give her a kiss"*

A special verb is the copula, *"be"* in English but not even realized by a verb in all languages. A copula can take different kinds of complement:

- an adjectival phrase: *"(John is) old"*
- an adverb: *"(John is) here"*
- a noun phrase: *"(John is) a man"*

The `Adjective` module

How to construct `AP`s. The main ways are

- positive forms of adjectives: *"old"*
- comparative forms with object of comparison: *"older than John"*

The `Adverb` module

How to construct `Adv`s. The main ways are

- from adjectives: *"slowly"*
- as prepositional phrases: *"in the car"*

1.3.7 Multimodality

The resource grammar library can also handle multimodal input in the form of general "pointing gestures". The idea is to extend the grammars with the single category `Point`. The library doesn't define the point category further, but leaves it to the specific domain implementation. As an example, in the multimodal MP3 player DJGoDiS, the user can click on artists, songs, and generic buttons displayed on the screen. Another example, in a multimodal tram information system tram stops are the interesting points.

The linearization type of points is a single string, with the special label point to make it possible to extend other linearization types:

```
lincat Point = {point : Str};
```

The linearization type of each grammatical category, such as NP, VP, S, QS and Utt, is then extended with the point linearization type:

```
lincat NP = Lang.NP ** {point : Str};
```

Here `Lang.NP` is the unimodal linearization type for noun phrases, which depends on the language. Finally, each function from the unimodal grammar library is extended to handle pointing gestures:

```
lin AdvNP np adv = Lang.AdvNP np adv ** {point = np.point ++ adv.point};
```

The `AdvNP` function adds an adverbial phrase to a noun phrase. The reference `Lang.AdvNP` is to the original unimodal function, and the result is extended with the multimodal input pointing gestures in sequence.

Some specific multimodal demonstratives such as "this", "that", "here", etc. are also defined, which takes a pointing gesture as argument:

```
fun this_point_NP : Point -> NP;
lin this_point_NP p = Lang.this_NP ** p;
```

These are in contrast with the original unimodal demonstratives, which have no associated point attached:

```
lin this_NP = Lang.this_NP ** {point = []};
```

Chapter 2

Generating Multilingual Grammars from OWL Ontologies in MIMUS

2.1 Introduction

MIMUS is a user-centered multimodal and multilingual dialogue system for the smart house domain; an intelligent, pro-active and voice-activated virtual butler who handles the full range of multimodal input and presentation possibilities for a spoken and click-based interface (see figure 2.1). It has been developed at the University of Seville as part of the TALK Project. As its immediate predecessor Delfos, it is based on the ISU approach. The keywords that best define the system are:

- Agent-based
- Intelligent
- User-centered
- Industry-oriented
- Fully multimodal for both input and output
- Ontology-based

It is therefore a context-aware and non-menu based system that fully adapts to the user needs according the user model configuration. The first version of the system is focused on the Smart Home scenario, and specifically oriented towards wheel-chair bound users' needs, although the general results can be extrapolated to other user types.

MIMUS incorporates a series of research results obtained during the project, such as:

- Grammar generation from ontologies
- Integration of OWL Ontological knowledge with ISU approach
- Dynamic reconfiguration of the home ontology

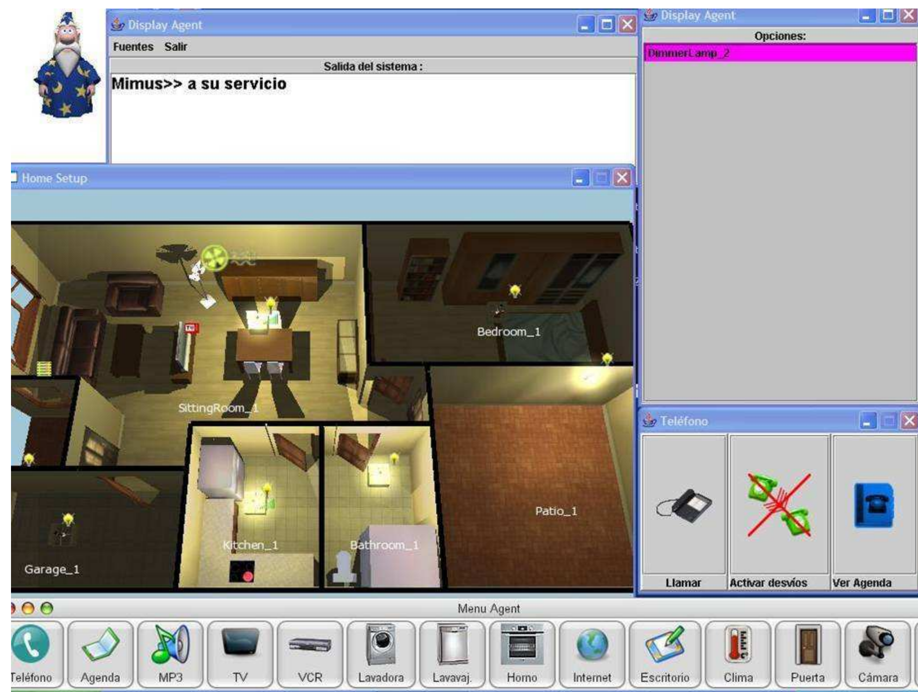


Figure 2.1: MIMUS Screenshot

- Multimodal Turn Planning
- Modality specific resources (3D house and talking head)
- Multimodal I/O fussion/fission
- Multimodal and multilingual grammar libraries
- Multilinguality on the fly
- It is based on a corpus of multimodal WoZ experiments with potential users in the in-home domain
- Non-menu based, pro-active system
- Speech-activated dialogue manager

2.2 Automatic grammar generation

The problem of manually generating grammars for a Natural Language Understanding (NLU) system has been widely discussed, and several authors have proposed different solutions. Two main approaches can be highlighted: Grammatical Inference and Rule Based Grammar Generation.
The Grammatical Inference approach[1] refers to the process of learning grammars and languages from data and is considered nowadays as an independent research area within the Machine Learning techniques.

[1] http://eurise.univ-st-etienne.fr/gi/

Examples of applications based on this approach are ABL (Zaanen, 2000) and EMILE (Williem Adriaans, 1992).

On the other hand, the Rule Based approach tries to generate the grammar rules from scratch (therefore based on the expertise of a linguist), but trying to minimize the manual work. An example of this approach is the Grammatical Framework (Ranta, 2004), whose proposal is to organize the multilingual grammar construction in two building blocks: the category and function declarations (abstract syntax) and the linearization rules (concrete syntax). The category and function declarations are done once and for all (shared by all languages), but the linearization rules are defined on a per-language basis. Methods which generate grammars from ontologies (including ours) are also examples of a Rule Based approach.

2.2.1 Generating grammars from ontologies

The separation of the Knowledge Manager (module in charge of the domain knowledge) and the NLU module of a Dialogue Manager system has a number of advantages such as reducing the complexity of linguistic components, reuse of existing domain knowledge, helping the dialogue manager on reference resolution (i.e: anaphoric expressions, underspecification, presupposition, and quantification), or helping the dialogue manager to keep dialogue coherence (Quesada and Amores, 2002; Milward and Beverige, 2003).

This Knowledge Manager module however has somewhat redundant information with the NLU module. The key idea of our work is that this redundancy can be used to automatically generate grammar rules from the relationships between the concepts described in the ontology. Thus, the fact that the concept **Lamp** is linked to the concept **Blue** through the **hasColor** relationship somehow implies that phrases like *the blue lamp* should be correct in this Domain and therefore accepted by the NLU grammar.

The generation of linguistic knowledge from ontologies has been previously proposed. In Russ et al. (1999) the authors proposed a method for generating context-free grammar rules from JFACC ontologies. Their approach was based on including annotations all along the ontology indicating how to generate each rule. They implemented a program that was able to parse the ontology and produce the grammar rules.

A second precedent of linguistic generation from ontologies can be found in Estival et al. (2004), where the author claimed that the concepts of an OWL ontology could be used to generate the lexicon of the NLU module.

In our work, a new rule-based solution for generating grammars from ontologies will be described. Section 2 motivates and gives an overview of the solution hereby proposed. Section 3 describes how the configuration files have to be built. Section 4 shows an introduction to the algorithm used. Section 5 includes real showcases of the tool at work. Section 6 is a summary of the conclusions and future work.

2.3 Solution overview

The solution proposed here is close to that of Russ et al. (1999) in the sense that we also parse the ontology for the rule generation. Nonetheless, it differs in two ways:

Firstly, the new approach argues that the ontology should remain as-is, without specific linguistic annotation. Although it is obvious that the ontology itself is not descriptive enough to generate the grammar rules without further information, it is preferable to place this additional information in a separate configuration file that describes how the grammar rules should be generated. This approach is more convenient for a

dialogue system (where the linguistic information in the ontology will be useless and cumbersome) and more suitable from a reusability point of view.
Secondly, OWL was chosen instead of JFACC mainly for two reasons:

1. The use of OWL is widely spread and seems to be the basis for the future semantic web. This implies that large ontologies are likely to be available in the future, and this approach will help create dialogue applications more easily by simply downloading specific domain ontologies.

2. OWL is based on RDF, and therefore uses Subject–Property–Object triplets. This static structure of OWL is of great help because the algorithm can focus on handling properties, letting the linguist define how to create rules that apply to all the elements in its Domain (or Range). Note therefore, that this choice is not just a change in the ontology format: the whole parsing algorithm is based on the RDF predefined structure.

As previously mentioned, this approach is completely focused on grammar rules generation: no automatic lexicon hierarchy generation has been considered. To ensure coherence between the lexicon and the grammar, a list of potential non-terminal types is extracted, that is, a list of all the entities within the ontology. The linguist decides which entities shall remain in the final dialogue application, and organizes them as Input-Forms (Amores and Quesada, 1997).
It is important to notice that this approach is meant only as a way to help the linguist, and will not provide a ready-to-use grammar. By using this tool, the grammar will be easier to generate and more consistent with the domain knowledge, but, in any case, the resultant grammar must be checked and completed manually in a second step.
The current implementation of this tool provides grammar rules in a format we have developed (Amores and Quesada, 1997), which basically consists of a left-hand symbol followed by an arrow and a list of right-hand symbols. Obviously this notation is translatable to most standards, like BNF.

2.4 Configuration files

As outlined above, the linguist must define a configuration file that will be used in conjunction with the ontology in order to generate the grammar rules. In this configuration file, the linguist has to identify the properties that may appear in the grammar and the way in which their domain and range will be included in the associated rules. In order to do it, an easy XML syntax has been defined (see DTD in figure 2.2).
Basically, the linguist can define the generation rules by means of nested *forEach* loops handling the properties (and subproperties) of the ontology, and using variables to identify the elements from its domain and range.
In order to better understand this structure as well as the objective of the tool, a selection of showcases including the relevant parts of the ontology, the configuration file and the resulting grammar rules are shown in the following sections.

2.5 Overview of the algorithm

In order to better illustrate how the algorithm works, this section will describe in more detail its functions. The algorithm consists of three major steps:

```
<!DOCTYPE rulesList [
  <!ELEMENT rulesList (forEach+)>
  <!ELEMENT forEach (forEach|rule+)>
  <!ELEMENT rule (left,right)>
  <!ELEMENT left (#PCDATA)>
  <!ELEMENT right (#PCDATA)>

  <!ATTLIST forEach property CDATA #IMPLIED>
  <!ATTLIST forEach subPropertyOf CDATA #IMPLIED>
  <!ATTLIST forEach domain CDATA #IMPLIED>
  <!ATTLIST forEach range CDATA #IMPLIED>

  <!ATTLIST rule lang (ES|EN|GR) #REQUIRED>
]>
```

Figure 2.2: DTD for the configuration file

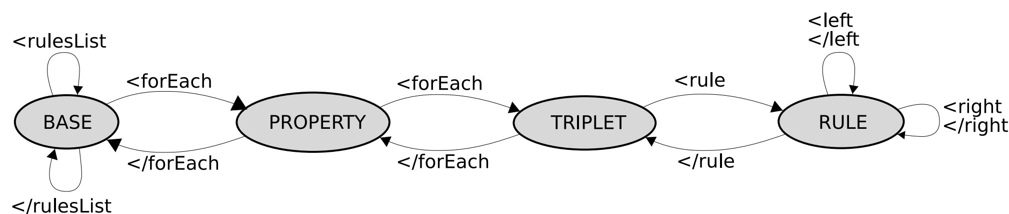


Figure 2.3: FSM for the configuration file parser

1. Parse the OWL ontology. The goal of this parsing is to generate an internal representation of the relevant ontological elements. This representation will in turn be used to make queries over the ontology.

2. Parse the configuration file. The objective here is to generate the list of all applicable rules.

3. Generate the output of rules. In this step, the script goes through the previous list of applicable rules, substituting the reference to classes and properties by the corresponding Input Form from the ontology.

The first two steps described have been implemented by a finite state machine (FSM) illustrated in figure 2.3. For each state in the FSM, only one set of attributes can be parsed. These are mentioned in the previous DTD structure:

Base :

- No attributes are expected in this state.

Property :

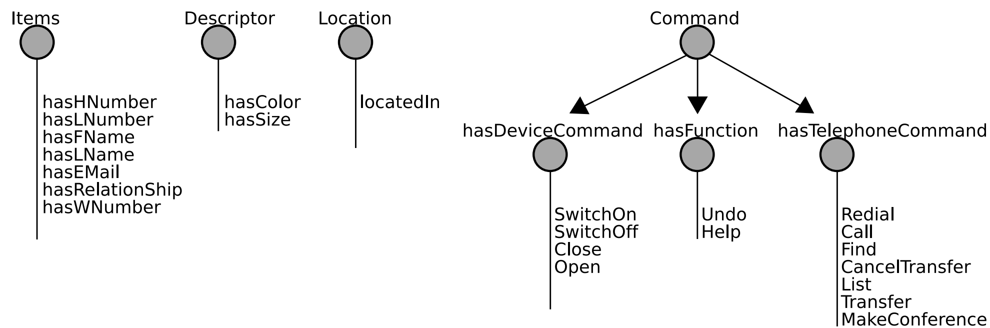


Figure 2.4: Ontology Structure

- propertyRef: Indicates the word that references the property in the rule description
- subPropertyOf: Indicates a superproperty. All the subproperties of this (including the indicated one) will be treated by the algorithm.

Triplet :

- domainRef: Indicates the word that references the domain in the rule description.
- rangeRef: Indicates the word that references the range in the rule description.

Rule :

- lang: Indicates what language the rule is valid for.

2.6 Showcases

2.6.1 Sample rules

The example below illustrates a common case in which the grammar rules will be generated. Our examples are taken from a smart-house domain in which the ontology describes both the hierarchy of devices in the house as well as the actions (or commands) which can be performed over those devices, such as *switch on the lamp in the kitchen*. Thus, consider an ontology where a set of properties are grouped as subproperties of a general **hasDeviceCommand** property. These properties are graphically displayed in figure 2.4.

In this showcase we are going to analyze the portion describing the device-related commands, whose XML equivalent is shown in figure 2.5. In this particular case, the linguist has detected that all properties are actually actions, that is, they correspond to the *commands* to be performed by the system over all the elements in the range, in this case, all devices within the ontology. This can be easily expressed by the configuration file in figure 2.6. Now, once the application is run indicating the appropriate configuration file, the following results are obtained:

Command → SwitchOff Fan
Command → SwitchOff Heater

```
<owl:ObjectProperty rdf:ID="SwitchOff">
   <rdfs:subPropertyOf rdf:resource="#hasDeviceCommand"/>
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Fan"/>
            <owl:Class rdf:about="#Heater"/>
            <owl:Class rdf:about="#Lamp"/>
            <owl:Class rdf:about="#Radio"/>
            <owl:Class rdf:about="#TV"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:range>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="SwitchOn">
   <rdfs:subPropertyOf rdf:resource="#hasDeviceCommand"/>
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Fan"/>
            <owl:Class rdf:about="#Heater"/>
            <owl:Class rdf:about="#Lamp"/>
            <owl:Class rdf:about="#Radio"/>
            <owl:Class rdf:about="#TV"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:range>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="Close">
   <rdfs:subPropertyOf rdf:resource="#hasDeviceCommand"/>
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range rdf:resource="#Blind"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="Open">
   <rdfs:subPropertyOf rdf:resource="#hasDeviceCommand"/>
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range rdf:resource="#Blind"/>
</owl:ObjectProperty>
```

Figure 2.5: Device-related commands in OWL XML syntax

```
<rulesList>
  <forEach property="P" subPropertyOf="hasDeviceCommand">
    <forEach domain="X" range="Y">
      <rule lang="ES">
        <left>Command</left>
        <right>P Y</right>
      </rule>
    </forEach>
  </forEach>
</rulesList>
```

Figure 2.6: Configuration file for device-related commands

Command → SwitchOff Lamp
Command → SwitchOff DimmerLamp
Command → SwitchOff Radio
Command → SwitchOff TV
Command → SwitchOn Fan
Command → SwitchOn Heater
Command → SwitchOn Lamp
Command → SwitchOn DimmerLamp
Command → SwitchOn Radio
Command → SwitchOn TV
Command → Close Blind
Command → Open Blind

It is important to note that even with this toy ontology, sixteen grammar rules have been generated using just two nested *forEach* loops.

2.6.2 Capturing multimodality

Now let us assume the same scenario (i.e. the same ontology) but including multimodal entries; namely voice and pen inputs. Following Oviatt's results (Sharon Oviatt and Kuhn, 1997), it may be expected that the mixed input modalities (voice: *switch this on*, pen: click on the lamp icon) may also include alternative constituent orders, that is, different to the voice only input. The NLU module may therefore receive inputs such as: *lamp switch on* (verb at the end).[2]
This new set of rules can be easily accounted for by adding just one rule to the configuration file, as shown in figure 2.7. The new output will be the same as before but including these new rules:

Command → Fan SwitchOff
Command → Heater SwitchOff

[2]Note that linguistically speaking this order is also possible in English in topicalized or left-dislocated constructions such as *The lamp, switch it on*.

```
<rulesList>
   <forEach property="P" subPropertyOf="hasDeviceCommand">
      <forEach domain="X" range="Y">
         <rule>
            <left>Command</left>
            <right>P Y</right>
         </rule>
         <rule>
            <left>Command</left>
            <right>Y P</right>
         </rule>
      </forEach>
   </forEach>
</rulesList>
```

Figure 2.7: Multimodal configuration file

Command → Lamp SwitchOff
Command → DimmerLamp SwitchOff
Command → Radio SwitchOff
Command → TV SwitchOff
Command → Fan SwitchOn
Command → Heater SwitchOn
Command → Lamp SwitchOn
Command → DimmerLamp SwitchOn
Command → Radio SwitchOn
Command → TV SwitchOn
Command → Blind Close
Command → Blind Open

2.6.3 Capturing multilinguality

Due to the structural differences among the human languages, different rules must be generated for different languages. For example, to indicate the location of a given device, it would be *the kitchen light* in English, whereas in Spanish the sentence order changes: *la luz de la cocina* (the light of the kitchen). Once the target language has been chosen, specific language rules must be generated.

Consider then the fragment in figure 2.8 taken from the ontology previously shown, describing which elements can be affected by the property **locatedIn**. The multilingual configuration file that would capture the structural differences mentioned above is shown in figure 2.9. Now, if only English grammar rules are to be generated, the application must be run with the option "-lang=EN", obtaining the following result:

Lamp → Bedroom Lamp
Lamp → Kitchen Lamp

```
<owl:ObjectProperty rdf:ID="locatedIn">
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Lamp"/>
                <owl:Class rdf:about="#Radio"/>
                <owl:Class rdf:about="#Heater"/>
             </owl:unionOf>
         </owl:Class>
     </rdfs:domain>
     <rdfs:range>
        <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Bedroom"/>
            <owl:Class rdf:about="#Kitchen"/>
            <owl:Class rdf:about="#Hall"/>
            <owl:Class rdf:about="#LivingRoom"/>
         </owl:unionOf>
        </owl:Class>
      </rdfs:range>
</owl:ObjectProperty>
```

Figure 2.8: The locatedIn property in OWL XML syntax

```
<rulesList>
   <forEach property="P" subPropertyOf="Location">
      <forEach domain="X" range="Y">
         <rule lang="ES">
            <left>X</left>
            <right>X P Y</right>
         </rule>
        <rule lang="EN">
            <left>X</left>
            <right>Y X</right>
        </rule>
      </forEach>
   </forEach>
</rulesList>
```

Figure 2.9: Multilingual configuration file

Lamp → Hall Lamp
Lamp → LivingRoom Lamp
Radio → Bedroom Radio
Radio → Kitchen Radio
Radio → Hall Radio
Radio → LivingRoom Radio
Heater → Bedroom Heater
Heater → Kitchen Heater
Heater → Hall Heater
Heater → LivingRoom Heater

2.7 Conclusions and future work

In this work a new Rule-Based approach to automatic grammar generation has been described. The proposed solution is based on OWL ontologies and provides linguists with an easy way to take advantage of the information contained within these ontologies. This information extraction process will also be easier for the linguist if the ontology has been designed keeping in mind that grammars will be generated from it.

The solution proposed has achieved the expected goals: the linguist can generate a good number of rules from a simple configuration files and, by having the rules directly generated from the ontologies, domain knowledge and linguistic knowledge coherence and completeness is ensured. In addition a rapid prototyping of new grammars for the speech recognizer and the NLU module is obtained by the same mechanism.

Future research areas include the generation of unification-based grammar rules, dialogue rules and evaluating the usefulness of the tool with very large OWL ontologies.

Chapter 3

Ontologies and Grammatical Framework

3.1 Relating existing ontologies with abstract GF grammars

In this section we view GF as a language for describing ontologies. The abstract syntax in GF is a type-theoretical logical framework with dependent types and function definitions (Ranta, 2004). This expressive power makes it possible to describe general ontologies.

Recall that the three main components of OWL are classes, individuals and properties. These have natural counterparts in abstract GF syntax. General OWL restrictions are also possible to implement in GF.

3.1.1 Classes and individuals

Classes can be implemented as GF categories. The OWL classes A, B and C will then be written as:

```
cat A; B; C;
```

Individuals are functions without arguments, i.e. constants. The OWL individuals a, a', b and c will be written as:

```
fun a, a' : A;
    b : B;
    c : C;
```

Subclasses

Since each individual has to have exactly one categoy, there is no direct notion of subclasses or subcategories in GF. That B is a subclass of A is instead represented by a coercion function, saying that all B's also are A's:

```
fun B_sub_A : B -> A;
```

That a class is a subclass of two other classes is then implemented as two coercion functions. If furthermore, both superclasses are subclasses of the same class, it can be neccesary to use function definitions

for enforcing equalities between instances. E.g. suppose that B is also a subclass of A', and that both A and A' are subclasses of C. Then we must state that for each b:B, the result is the same regardless of whether we go via A or A':

```
fun B_to_C : C -> C;
def B_to_C (A_sub_C (B_sub_A x)) = (A'_sub_C (B_sub_A' x));
    B_to_C x = x;
```

Uninhabited superclasses

There is one common special case of subclasses, and that is when the superclass does not have any instances of its own. This superclass can then be implemented in GF as a dependent category. Suppose that both D, E and F are subclasses of the class Super, which itself does not have any instances. This can be written as:

```
cat Super DEF;
    DEF;
fun D, E, F : DEF;
```

This means that the subclass D corresponds to the GF (dependent) category Super D. An instance d of the subclass D is now written:

```
fun d : Super D;
```

Instances of several classes

It is not possible to directly state that an individual is an instance of two classes, since a GF term can only have one category. We solve this by introducing intersections of classes instead. That ab is an instance of both A and B will then be written as:

```
cat AB;
fun AB_sub_A : AB -> A;
    AB_sub_B : AB -> B;
    ab : AB;
```

I.e. ab is an instance of the intersection of A and B, which is a subclass of both A and B.

Class equivalence

In OWL there are two notions of class equivalence – intensional and extensional. That two classes A and B are extensional equivalent simply says that they are subclasses of each other:

```
fun A_sub_B : A -> B;
    B_sub_A : B -> A;
```

This equivalence can be further enforced by defining equivalence functions on A and B:[1]

```
fun equiv_A : A -> A;
def equiv (B_sub_A (A_sub_B x)) = x;
    equiv x = x;
```

Intensional equality is not directly possible in GF, however, since operator definitions are not allowed in abstract syntax.

3.1.2 Properties

Properties can be implemented as categories which depend on the domain and range classes. The OWL property Prop with domain A and range B is thus written as:

```
cat Prop A B;
```

An individual a which has the property Prop(b) will be a GF constant:

```
fun a_Prop_b : Prop a b;
```

Functional properties

A special case is when a property Qrop is functional, i.e. that each element in the domain is connected to one and only one element. This can be implemented in GF by using function definitions. The property itself is represented by a function from A's to B's:

```
fun Qrop : A -> B;
```

Each instance of the property is then a row in the definition of Qrop:

```
def Qrop a  = b;
    Qrop a' = b';
```

Alternative view of properties

There is an alternative way of implementing general properties, which is inspired from the previous idea. We can declare the category ListB, with its constructor functions BaseB and ConsB:

```
cat ListB;
fun BaseB : ListB;
    ConsB : B -> ListB -> ListB;
```

Now the general property Prop can be implemented as a function from A to ListB:

[1] The definition of `equiv_B` is analoguous.

```
fun Prop : A -> ListB;
```

The facts that hat a is not related to anything, and that a' is related to both b and b', are written as:

```
def Prop a  = BaseB;
    Prop a' = ListB b (ListB b' BaseB);
```

There is a syntactic sugar in GF for writing lists. The category ListB can be written [B], and the declarations above are then written:

```
cat [B]{0};
fun Prop : A -> [B];
```

The {0} in the declaration of [B] says that BaseB takes no arguments, i.e. that [B] can be uninhabited.

Cardinality restrictions on properties

OWL cardinality restrictions can be implemented in the same way. That each A is related to at least three B's can be declared as:

```
cat [B]{3};
fun Prop : A -> [B];
```

Maximality restrictions can also be implemented, albeit cumbersome. That each A is related to at least one and at most three B's can e.g. be declared as:

```
cat B123;
fun B1 : B -> B123;
    B2 : B -> B -> B123;
    B3 : B -> B -> B -> B123;
fun Prop : A -> B123;
```

Subproperties

In OWL, a property can be declared to be a subproperty of another. Following the idea from subclasses, we can either use a coercion function, or dependent categories.

A very common case is when the superproperty does not have any direct instances, in which case we can declare it as depending on the subproperties. Suppose both Qrop and Trop are subproperties, with domain A and range B, of Prop which doesn't have any direct instances, we write this as:

```
cat Prop QT A B;
    QT;
fun Qrop, Trop : QT;
```

To say that a is related by Qrop to b and by Trop to c, we write:

```
fun a_Qrop_b : Prop Qrop a b;
    a_Trop_c : Prop Trop a c;
```

When the superproperty can be inhabited, we need a coercion function. Supposing that Qrop is a subproperty of Prop, we declare a coercion function from Qrop x y to Prop x y:

```
fun Qrop_sub_Prop : (x:A) -> (y:B) -> Qrop x y -> Prop x y;
```

The most general case is when the subproperty's domain and range are subclasses of the superproperty's domain and range. In this case we get the following:

```
fun Qrop_sub_Prop : (x:A) -> (y:B) -> Qrop x y
                 -> Prop (A_sub_A' x) (B_sub_B' y);
```

Here A_sub_A' and B_sub_B' are the coercion functions between the domains and ranges. There are even more possibilities for implementing subproperties, e.g. if the superproperty's domain or range is uninhabited, in which case this could be implemented as dependent categories.

Datatype properties

GF has primitive notions of strings, integers and real numbers, with the categories String, Int and Real. This means that datatype properties which ranges over strings, integers or reals are straightforward to handle.

3.1.3 General OWL axioms and restrictions

More general axioms and restrictions on an ontology can be difficult to implement in GF. One example is negated propositions, e.g. stating that two classes are disjoint. GF has no primitive notion of negation, but uses the type-theoretical definition that negation means implication of absurdity (Nordström et al., 1990).
To handle general OWL restrictions we can always resort to the standard solution in type-theory, to declare a category of logical propositions, in parallel with the category of proofs of propositions:

```
cat Prop;
    Proof Prop;
```

We refer to e.g. Nordström et al. (1990) or Ranta (1994) for more discussion about these topics.

3.1.4 Ontologies and concrete syntax

Annotation properties in OWL are for non-ontological information about classes and/or individuals. There are some predefined annotation properties, e.g. rdfs:comment, rdfs:seeAlso and rdfs:label. For our purposes the most interesting is rdfs:label:

> rdfs:label is an instance of rdf:Property that may be used to provide a human-readable version of a resource's name. (...) Multilingual labels are supported using the language tagging facility of RDF literals. (Brickley and Guha, 2004, section 3.6)

This is a description of a very simple version of the concrete syntax of a GF grammar, where the language tag says which concrete syntax the label is for. This means that all label annotations for a certain language in an ontology can be straightforwardly instantiated as a GF concrete syntax.

GF concrete syntax is much more expressive than that, since it supports function arguments and complex datatypes.

3.2 Incorporating the Mimus in-home ontology

In this section we show as an example how the Mimus ontology described in chapter 2 can be implemented as the abstract syntax of a GF grammar. The XML configuration files that are used to generate context-free recognition grammars then have their natural interpretations as concrete syntaxes of the GF grammar.

3.2.1 The Mimus ontology as an abstract GF grammar

When doing the conversion from the Mimus ontology, we chose to implement all subclasses and subproperties with coercion functions as explained before. An alternative which we didn't pursue would be to use dependent categories for implementing the device classes and the command properties.

The dialogue system

There is a class System in the ontology, which (commonly) has one element – the representation of the particular domain.

```
cat System;
fun mimus : System;
```

Devices

There are six devices which are all subclasses of the Device class.

```
cat Device;
    Blind; Fan; Heater; Lamp; Radio; TV;
```

Some of the commands can operate on a number of devices. In the ontology this is done by making the domain the union of these classes, as is shown in the XML code in figure 2.5 on page 20. In GF the union has to be given a name, and it must be stated that each class is a subclass of the union.

```
cat FanHeaterLampRadioTV;
fun Fan_sub_FHLRT : Fan -> FanHeaterLampRadioTV;

    TV_sub_FHLRT  : TV  -> FanHeaterLampRadioTV;
```

Each device is also a subclass of the Device class. We optimize this a bit by stating that the union is a subclass, instead of having to state that each device is a subclass.

```
fun Blind_sub_Device : Blind -> Device;
    FHLRT_sub_Device : FanHeaterLampRadioTV -> Device;
```

Command properties

The command properties Open and Close have a System as domain and operates on a Blind.

```
cat Open  System Blind;
    Close System Blind;
```

The command properties SwitchOn and SwitchOff also have a System as domain, but can operate on any Fan, Heater, Lamp, Radio or TV.

```
cat SwitchOn  System FanHeaterLampRadioTV;
    SwitchOff System FanHeaterLampRadioTV;
```

Now each of these four properties is a subproperty of HasDeviceCommand, which operates on Devices in general.

```
cat HasDeviceCommand System Device;
```

We have to state that any Open (and Close) command is a HasDeviceCommand, and that each Blind also is a Device, by applying the coercion Blind_sub_Device.

```
fun Open_sub_Cmd  : (sys:System) -> (dev:Device)
                  -> Open sys dev
                  -> HasDeviceCommand sys (Blind_sub_Device dev);
    Close_sub_Cmd : ...
```

The same goes for SwitchOn and SwitchOff, where we have to say that any of their devices also is a Device.

```
fun SwitchOn_sub_Cmd  : (sys:System) -> (dev:FanHeaterLampRadioTV)
                      -> SwitchOn sys dev
                      -> HasDeviceCommand sys (FHLRT_sub_Device dev);
    SwitchOff_sub_Cmd : ...
```

3.2.2 Mimus configuration files as GF concrete syntax

Each language in the configuration file becomes a specific concrete syntax in the GF grammar.

Linearization of commands

Recalling the configuration file for extracting commands to a recognition grammar, as shown in figure 2.6 on page 21, we can implement this as one single GF function taking a System, a Device and a command, returning a Command.

```
fun command : (sys:System) -> (dev:Device)
            -> HasDeviceCommand sys dev -> Command;
```

Since each command property is a subproperty of HasDeviceCommand, they will match this rule. The main part of the configuration file comes in the linearization of this function. The configuration stated that the command was the command name followed by the device, which is specified in GF as:

```
pattern command sys dev cmd = cmd ++ dev;
```

Linearization of subclass and subproperty coercions

Each function that is defined in the abstract syntax needs a concrete instantiation too. The ones missing for us now are the subclass and subproperty coercion functions, which in GF simply become identity functions.

```
pattern Fan_sub_FHLRT    x = x;

        FHLRT_sub_Device x = x;
```

Subproperty linearizations have some dummy arguments for the domain and the range, which can be ignored.

```
pattern Open_sub_Cmd      _ _ x = x;

        SwitchOff_sub_Cmd _ _ x = x;
```

3.2.3 Discussion

In this section we have shown that an ontology can be converted to a GF grammar, for which we then can write multilingual recognition grammars. The ideas are similar to the XML configuration files presented in section 2.4.

Advantages with using GF

The main advantage with converting the ontology to GF is that we can make use of the rich type system in the concrete syntax, for capturing e.g. inflectional patterns or discontinuous constituents.

One example is descriptive properties, such as hasColor and hasSize, linking colors and sizes to devices. The idea is to make it possible to say e.g. “the blue lamp” or “the small TV”.

In an English grammar we can still use a simple context-free grammar of the form:

Lamp → Size Lamp
TV → Size TV
Lamp → Color Lamp
TV → Color TV

However, in a language such as German where the adjective agrees with the gender of the noun, this is not correct. This problem can be solved by using more complex linearization types in GF. It is enough to state that adjectives are tables that depend on a gender, and nouns have an inherent gender.

We can even go one step further and make use of the resource grammar described in section 1.3, which gives us a multilingual grammar automatically.

Disdvantages with using GF

The main disadvantage with converting the Mimus ontology to GF is the same as for any OWL ontology – subclass coercions become quite cumbersome, and general restrictions can be very complicated in GF. But since the conversion from the Mimus ontology to a recognition grammar does not make crucial use of restrictions this is not a big problem.

3.3 GF abstract syntax as OWL ontologies

3.3.1 Context-free grammars as OWL ontologies

A context-free grammar can be seen as an ontology. This is known previously, especially in computer science texts for compiler construction and parsing, see e.g. Appel (1998):

- Each category A is a class in the ontology – i.e. the class of syntax trees with mother node A
- Each rule $A \rightarrow X_1 \cdots X_n$ is a subclass of A – i.e. the class of syntax trees constructed from that rule

Note that the individuals of a class A are syntax trees, but not necessarily all trees. We have to construct each tree in the ontology as an individual.

Subtrees as properties

Given a rule/subclass $A \rightarrow X_1 \cdots X_n$ (call it R), we define n daughter properties **arg-1**, ..., **arg-n**:

- A daughter category $B = X_i$ of R yields a property **arg-i** from R to B
- A terminal token $t = X_i$ of R yields a datatype property **arg-i** R to a string

Since the right-hand side of a rule is ordered, the daughter properties need to be ordered too. Therefore we name the properties $arg_1, \ldots, arg_n$.
If we want to formally implement a context-free grammar as an OWL ontology, the daughter properties have to be unique for each rule/subclass R. Therefore the formal name of property **arg-i** should be **arg-i–A-X1-...-Xn** or something similar. This means that the ontology will have an very large number of properties (and long property names).

3.3.2 GF abstract syntax in OWL

The abstract syntax of a GF grammar is a context-free grammar, but without terminal symbols and where each rule has a unique name. This simplifies the implementation as an OWL ontology:

- Each category A corresponds to a class A
- Each rule $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow A$ corresponds to a subclass f of A
- Each daughter category A_i of f corresponds to a property $P[A_i]$ from f to A_i

The resulting ontology is simpler than the ontology for the context-free grammar in the following senses:

- All datatype properties have disappeared, since there are no terminal symbols
- The names of the rule classes are shorter, since the GF rules have unique names
- The order between the daughters $A_1, \ldots, A_n$ is not of real importance in GF – it is the concrete syntax that is ordered, not the abstract. Therefore we do not have to supply the exact index of the argument, it is enough with the name of category; e.g. given the rule f : **Det** → **Noun** → **NP**, the property P[**Noun**] from f to **Noun** can be called simply **noun**, instead of **arg-2**
- A consequence of this is that there will be a great deal less properties in the ontology, since many rules can share the same property.

One problem is when a GF rule has more than one argument with the same category, e.g. a flat sentence generating rule:

```
fun s1 : NP -> Verb -> NP -> S;
```

In this case it is difficult to specify which argument NP's belong to the subject and object. This can be solved in two ways: either by introducing a new name for one of the property names, or by adding a new coercion category to the grammar:

```
cat ObjectNP;
fun objectNP : NP -> ObjectNP;
```

If we have this, we can change the type of `s1` and use the conversion we have already described:

```
    s1 : NP -> Verb -> ObjectNP -> S;
```

3.3.3 Optimizing the grammar ontology

In this section we present two possible optimizations for the translation of a GF grammar to an OWL ontology, which decreases the size of the ontology.

One-argument rules

Suppose that there is a rule `intrans` in the grammar taking only one argument:

```
fun intrans : Verb -> VP;
```

If there are no more rules with type `Verb -> VP`, then the rule name `intrans` can be inferred from the context. This means that it is not necessary to create the class **intrans** (being a subclass of **VP**) together with the property **verb** to the daughter class **Verb**, but we can instead state that **Verb** is a subclass of **VP** directly.

Note that this can be applied to the problem of multiple arguments with the same category described above. Since the coercion rule `objectNP` is a unique single-argument rule, we can simply state that **NP** is a subclass of **ObjectNP**.

Optional arguments

A context-free rule with optional arguments can be converted to one single OWL rule class, where the optional arguments become optional properties.

GF does not support optional arguments, which means that this must be implemented by several GF functions. Suppose that we have the following context-free rule with two optional arguments:

NP → Det (AP) Noun (PP)

This has to be simulated by four GF rules:

```
fun np0 : Det -> AP -> Noun -> PP -> NP;
    np1 : Det       -> Noun -> PP -> NP;
    np2 : Det -> AP -> Noun       -> NP;
    np3 : Det       -> Noun       -> NP;
```

And if we used our translation above, we would get four subclasses of the **NP** class. But these can be merged into one single subclass where the daughter properties **ap** and **pp** are optional, whereas **det** and **noun** are obligatory.

One problem remains, and that is the name of the merged class. It must be possible to recreate the original GF name by knowing the name of the class and which arguments that are instantiated. One solution is to enforce that the name of each GF rule has the same prefix, followed by a uniquifying suffix (in this case a number). No other rules in the grammar are allowed to start with the same prefix. The name of the merged class will be the common prefix, and then it is possible to infer which of the GF rules to use.

3.3.4 The GF resource grammar as an OWL ontology

The translation described in this section can be used on the multilingual GF resource grammar. But since the resource grammar has a very limited lexicon, we divide the ontology into two parts to facilitate the writing of the domain-specific lexicon:

- All grammatical categories (such as noun phrases, verb phrases, sentences and questions) are subclasses of the class **Phrase**
  - the grammar rules occur as subclasses of the grammatical categories
- All lexical categories (such as verbs, nouns, determiners and prepositions) are subclasses of the class **Lexicon**
  - the lexical entries occur as instances of the lexical categories

With this structure it is possible to automatically create the abstract syntax of a lexical GF grammar, which is imported by the domain grammar.

In theory it would also be possible to instantiate the lexicon for different concrete languages, by using the built-in annotation property **rdfs:label**. But this turns out to be very difficult for most languages, since the inflectional patterns are often irregular. It is possible to solve by more complicated properties and morphological class hierarchies for each language, but we do not pursue this issue further and instead resort to writing the concrete lexicon instances as GF grammars directly.

3.3.5 Ontology editing as a multilingual authoring system

Dymetman et al. (2000) noted that an ontology for abstract syntax can be used for multilingual document authoring. This suggests that our OWL interpretation of the resource grammar, any OWL ontology editor such as Protegé[2] can be used as a multilingual authoring system for generating grammatically correct sentences:

- OWL is used to create abstract syntax trees, ...
- ... which are then automatically translated to GF format, ...
- ... and linearized in all languages implemented in the resource grammar.

This idea is used in section 3.4.1 for specifying multilingual user and system utterances in a dialogue domain.
Note that the GF system itself comes with tools supporting multilingual document authoring (Khegai et al., 2003), which can be used as an alternative to an OWL editor.

3.4 Using ontologies to specify GoDiS dialogue domains

A whole dialogue domain for a GoDiS dialogue system can be specified as an OWL ontology. From such an ontology grammars for system utterances and user utterances are automatically generated, as well as GoDiS specification files.
The specification of the dialogue specific information, such as dialogue plans and sortal restrictions, and the generation of the GoDiS specification files, is presented in TALK deliverable D2.2 (Milward et al., 2006). In this section we only describe the parts that are used to specify the domain-specific utterances.

3.4.1 The GF resource grammar

The main idea is that the domain-specific utterances for the given dialogue domain are specified as syntax trees in the resource grammar ontology described in section 3.3.
The resource grammar is grouped by the top-level class **Syntax**, of which the grammar classes **Phrase** and **Lexicon** are subclasses. The grammatical categories are subclasses of **Phrase**, and the syntax trees are instances of these classes.

3.4.2 Actions and predicates

The *actions* in GoDiS are used when the user requests the system to perform something, e.g. to restart the dialogue, or give the user some help. In OWL, each GoDiS action becomes an instance of the toplevel class **Action**. In the example domain there are the following actions:

- **restart** (“start over please”)

[2] http://protege.stanford.edu/

- **help** (“please give me help”)

The 1-place *predicates* in GoDiS are used for forming wh-questions, and for forming answers to questions. Each GoDiS predicate becomes an OWL individual in the toplevel class **Predicate**. In the example domain there are the following predicates:

- **departure** (“from where do you want to go?” / “from the central station”)
- **destination** (“to where do you want to go?” / “to the central station”)
- **shortest_route** (“what is the shortest route between the two given stops?” / “the shortest route is ...”)

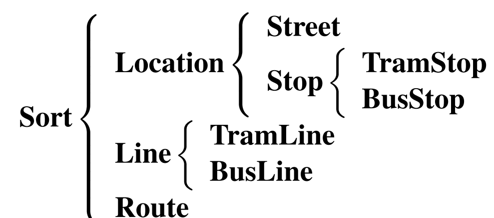
To the actions and predicates of a domain we associate utterances that the user and system can perform. In GoDiS these were originally defined in a Prolog predicate as a phrase spotting lexicon. However, in TALK deliverables D1.1 and D1.2 (Ljunglöf et al., 2005, 2006) we have shown how to implement the utterances in terms of multilingual and multimodal grammars in GF.
In the ontology we associate each utterance with a syntax tree from the GF resource grammar:

- For each **Action** there is an associated property **actionPhrase** which associates a verb phrase (**VP**) with the action.
- Each **Predicate** has two associated properties – a wh-question and an answer. Thus we have the two properties **questionPhrase** and **answerPhrase**, which associate a wh-question (**WhQ**) and an answer (**Clause**) to each predicate.

In the GF resource grammar these categories (**VP**, **WhQ** and **Clause**) are very general, i.e. they can be realized in several different linguistic forms. This gives the freedom to e.g. either form a request from the user (“start over please”), or some kind of feedback from the system (“do you want to start over?”, “I’m starting over”).

3.4.3 Sorts and indivuduals

The *sortal hierarchy* is an OWL subclass hierarchy under the top-level class **Sort**. Consider the following example hierarchy from the Tram domain:



This subclass hierarchy says that each **TramStop** is also a **Stop**, which is also a **Location**.

Connecting the sorts to the grammar

The individuals of each sort can be used either directly as *short answers*, or as arguments to predicates. Subsorts have to be of the same grammatical category as their mother sorts, which means that it is the direct subclasses of **Sort** that decides the associated grammatical category.

Thus, each direct subclass of **Sort** has a property mapping the instances to a syntax tree of the associated grammatical category. For the example hierarchy from the Tram domain, we get the following properties:

- **locationPhrase**, mapping each **Location** to a **ProperName**
- **linePhrase**, mapping each **Line** to a **ProperName**
- **routePhrase**, mapping each **Route** to a **Sentence**

A common case is when a sort is a database of entities, such as **Location** and **Line** above. In these cases the concrete syntax often consists of strings with very regular (or non-existent) morphology, and then the annotation property **rdfs:label** can be an alternative instead of syntax trees. The only thing to think about is that the sort in question has to be made a subclass of the class for the grammatical category in the grammar. Thus, **Location** and **Line** should be subclasses of **ProperName**.

Chapter 4

The Enhanced Multimodal Grammar Library

The GF grammar library consists of a domain-independent group of grammars and three domain-specific grammar groups. The domain-independent grammar group is called Common. Each domain-specific grammar group corresponds to a GoDiS application. The three domains are Agenda, MP3 and Tram which in turn correspond to the three GoDiS applications *AgendaTalk*, *DJGoDiS* and *TramGoDiS*. The grammar library is divided into a system grammar and a user grammar which share common modules.

The overall structure of the grammars for a specific domain is shown in figure 4.1. The domain-specific grammars separates the system and user utterances while sharing a common domain-independent grammar module with other domains. Every domain-specific grammar can have its own resources. The resource in the Agenda domain is called Bookings, the MP3 domain uses a resource called Music and the Tram domain has two resources, Stops and Lines.

To describe the grammars in the library, examples will be taken from the MP3 module. Both to illustrate the general theory, but also to give a more in-depth description of a domain-specific grammar module.

4.1 Separating system and user utterances

Throughout the GF grammar library the grammar modules are split into a system and a user part with shared additional resources.

From the perspective of a GoDiS dialogue system the system grammars are used for parsing the semantic representation in Prolog syntax of the dialogue moves into an abstract syntax that can be used for linearization into the natural language used by the user. The user grammars are correspondingly used for translating natural language utterances into a semantic representation in Prolog syntax, via the abstract syntax.

The separation into system and user grammars gives more efficient grammars, constructed for their specific needs. It is not necessary to have user linearizations for some system specific questions. The user will never ask the system “Do I want to add a song?”. However, the system has to be able to talk about everything within the system. If the confidence score on the received user input is low the system will try to ground a request before acting upon it, “Add a song, is that correct?”

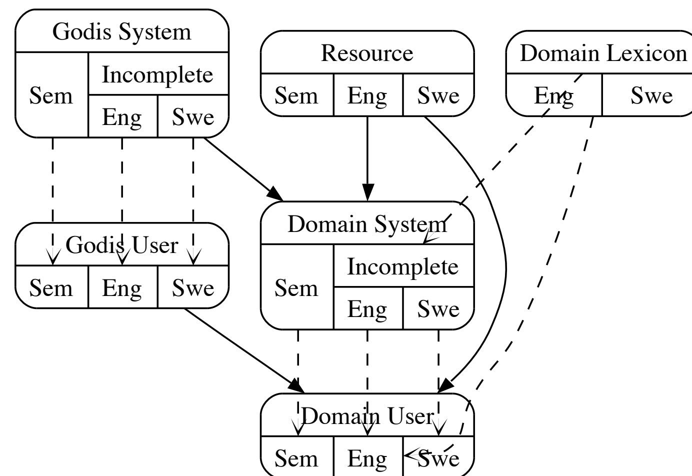


Figure 4.1: The structure of the grammars for a domain application

Another difference is that we want the user grammar to recognize all possible ways different users will pose questions, give answers or request actions. For the system we only need one way of phrasing an utterance.

Since the system has to be able to talk about everything, including user utterances, the system grammar includes a minimal recognition grammar for user utterances. Therefore, the user grammar can be seen both as a specialization and an extension of the system grammar – it specializes the grammar to only those utterances that the user will say, but it extends the number of alternative ways the utterances can be phrased. Another consequence is that a small and unefficient user grammar can be constructed straight from the system grammar.

4.2 The domain-independent grammars

All domain-independent grammars are inside the Common group of grammars. The common system grammar is called `GodisSystem` and specifies the two common features for all GoDiS applications. The first feature is utterances used for various dialogue moves such as reraising issues, greetings, feedback and grounding. The other feature is to define the internal structure of the dialogue moves from the categories Question, Action, Proposition and ShortAnswer. This can be seen as the common grammar specifying the upper level of the grammar structure, letting the domain-dependent grammars implementing the low-level details.

System utterances

The multilingual system grammar `GodisSystemI`[1] is implemented by using syntax trees from the GF resource grammar. This guarantees that all system utterances will be grammatically correct and can be

[1]The trailing `I` in the grammar name says that the grammar is *incomplete*, or multilingual, and can be instantiated by several different languages.

implemented in any language provided by the resource grammar. From this multilingual grammar, the concrete languages `GodisSystemEng` and `GodisSystemSwe` are automatically derived by instantiating with a suitable lexicon. The system grammar `GodisSystemSem` gives GoDiS semantics for each of the dialogue moves in the grammar.

User utterances

The user grammar is correspondingly called `GodisUser` and defines the relationship between the user-specific dialogue moves requesting, answering, asking and giving a short answer.

Resources

There are two common resources, one for the semantic implementations and one for implementing the natural language grammars. The semantic resource `Prolog` converts the abstract syntax trees into the Prolog annotation used by GoDiS to represent the semantics of the dialogue system. The natural language resource `GodisLang` is an interface between the GF resource grammar and the dialogue system grammars. It gives language-dependent linearizations for greetings and other general utterances. It also provides macros for transforming common phrases in a dialogue system into the syntax trees given by the GF resource grammar.

4.3 The domain-specific grammars for system utterances

The domain-specific grammars are just like the domain-independent grammars separated into a system grammar and a user grammar. In this section we describe the structure of the system grammar.

Abstract syntax

The system grammar has to include the following entities, where the examples are from the MP3 domain:

- Each sort is defined as a GF category, together with its instances:

  ```
  cat Artist; Song;
  fun abba, clash : Artist;
      happy_new_year, london_calling : Song;
  ```

  Commonly this information is defined in a separate grammar, such as the `Music` resource grammar for the MP3 domain.

- Each sort can also be used as a short answer:

  ```
  fun artist : Artist -> ShortAns;
      song   : Song   -> ShortAns;
  ```

- Each action is a constant of the category `Action`:

```
fun playlist_add : Action;
```

- Each predicate can be used in two different ways, either as a wh-question, or applied to an argument as a proposition:

```
fun song_to_play_Q : Question;
    song_to_play_P : Song -> Proposition;
```

This is all that has to be defined for a domain – the common GoDiS grammar knows how to use short answers, actions, questions and propositions to build all possible dialgoue moves.

Natural language syntax

The system utterances for natural languages are language independent. This is obtained by using an *incomplete* GF grammar with syntax trees from the GF resource grammar. Since many grammatical constructions occur repeatedly we use macros defined in `GodisLang` as an interface to the resource grammar. The categories for short answers, actions, questions and propositions have their linearization inherited from categories in the resource grammar:

- Each sort has its own linearization category, often being a noun phrase (`NP`):

```
lincat Artist = NP;
lin abba = plur_NP ["Abba"];
```

  The macro `plur_NP` is defined in `GodisLang` for creating a plural noun phrase from a string.

- Short answers are linearized as noun phrases (`NP`):

```
lincat ShortAns = NP;
lin artist x = x;
```

- Actions are linearized as verb phrases (`VP`), together with extra information about the `ClauseForm`:

```
lincat Action = ClauseForm ** VP;
lin playlist_add = hasDone **
       ComplV3 add_to_V3 (indef_N_sg song_N) (the_N_sg playlist_N);
```

  Actions are used in different contexts by the common GoDiS grammar, such as requests ("Add a song to the playlist please"), questions ("Do you want to add a song to the playlist?") or confirmations ("I have added a song to the playlist"). The information needed for all this is built into the GF resource grammar.

- Questions are linearized as question clauses (`QCl`), plus `ClauseForm` information:

```
lincat Question = ClauseForm ** QCl;
lin song_to_play_Q = isDoing ** which_N_do_you_want_to_V2 song_N play_V2;
```

Questions are also used in different contexts, such as questions ("Which song do you want to play?") or accomodation ("Returning to the issue about which song you want to play").

- Propositions are linearized as clauses (`Cl`), plus `ClauseForm` information:

```
lincat Proposition = ClauseForm ** Cl;
lin song_to_play_P x = isDoing ** you_want_to_VP (ComplV2 play_V2 x);
```

Propositions can be used in contexts such as answers and feedback ("You want to listen to London calling") or y/n-questions ("Do you want to listen to London calling?").

The linearization categories for short answers, actions, questions and propositions are declared in the common `GodisSystemI` grammar. The `ClauseForm` information tells in which form (e.g. present or past tense) the phrase should be linearized, in system reports. I.e. some actions should be reported in present tense ("I am playing the song"), whereas other should be in past tense ("I have added the song to the playlist").

GoDiS semantics

The system grammar for GoDiS semantics is relatively straightforward to implement by using the macros defined in the `Prolog` resource module:

- Short answers are one-place Prolog functors:

```
lin artist = pp1 "artist";
    song   = pp1 "song";
```

- Actions are Prolog atoms:

```
lin playlist_add  = pp0 "playlist_add";
```

- Predicates are either one-place predicates, or wh-questions of the form `X^p(X)`:

```
lin song_to_play_P = pp1  "song_to_play";
    song_to_play_Q = pWhQ "song_to_play";
```

Obviously, there will be duplicated information in the semantics file – the GF functions commonly have the same name as the corresponding GoDiS terms. This suggests that the semantics grammar is fairly simple to automatically generate.

4.4 The domain-specific grammars for user utterances

There are primarily four kinds of domain-specific dialogue moves the user can perform:

Dialogue move	English utterance	GoDiS semantics
Giving a short answer	*"The Clash"*	answer(artist(clash))
Requesting an action	*"Add a song"*	request(playlist_add)
Asking a question	*"Which is the current song?"*	ask(X^current_song(X))
Answering a question	*"It is London calling I want to play"*	answer(song_to_play(london_calling))

The common grammar for user utterances, GodisUser, defines the four categories ShortAns, Action, Question and Answer. Note that these categories are distinct from the corresponding categories in the system grammars. A typical utterance by the user often also gives extra information in addition to the main dialogue move. This extra information has in GoDiS the form of answers or short answers:

English utterance	GoDiS semantics
" London calling by the Clash"	answer(song(london_calling)) + answer(artist(clash))
"Add London calling by the Clash"	request(playlist_add) + answer(song_to_add(london_calling)) + answer(artist_to_add(clash))
"What songs have the Clash made?"	ask(X^available_song_Q(X)) + answer(artist(clash))

The linearization types for the dialogue move categories in the user grammar are simple strings, not complex phrases from the resource grammar. The main reason for this is that the user utterances will only be used in one way, since the system replies are already defined in the system grammar. Another reason is that some user utterances that we want to capture can be grammatically incorrect and therefore not covered by the resource grammar. A third reason is that it should be simple to add a new synonym for the same semantics. These new synonyms can e.g. be taken from a corpus of user utterances, and then it is easiest to just add a string instead of a syntax tree.

Abstract syntax

The abstract syntax consists of one rule for all synonymic utterances. Two utterances are synonyms if they are interpreted in the same way by GoDiS, i.e. they have the same translation into a list of GoDiS dialogue moves. This means that any sort, action or predicate that the user would want to talk about will have a user-specific variant:

- Giving a short answer:

```
fun artist : Artist -> ShortAns;
```

- Requesting an action:

```
fun playlist_add : Action;
```

- Asking a question:

```
fun current_song : Question;
```

- Answering a question:

```
fun song_to_play : Answer;
```

Note that these functions look very similar, some of them even identical, to the corresponding rules in the system grammar. There are two important differences, however.

The first difference is that not all rules from the system grammar will be in the user grammar. This is true for most predicates, which will either have the question form or the answer form in the user grammar, not both forms. E.g. since it is the user who asks which song is the current one, it is very unlikely that the user gives an answer to that question.

The second difference is that the user can give extra information in the form of additional answer moves. This means that the user grammar will contain rules taking additional arguments:

- Combined answers, e.g. *"London calling by the Clash"*:

```
fun song_artist : Song -> Artist -> ShortAns;
```

- Combined actions, e.g. *"Add London calling by the Clash"*:

```
fun playlist_add__song_artist : Song -> Artist -> Action;
```

- Combined questions, e.g. *"What songs have the Clash made?"*:

```
fun available_song__artist : Artist -> Question;
```

Natural language syntax

The linearization types for user utterances are simple strings, which means that it is possible to just list all synonym strings for an utterance:

```
lin current_song = {s = variants{
        ["what is the name of the current song"];
        ["what is the name of this song"];
        ["which is the current song"];
        ["which song is this"]}};
```

The first two strings can also be optimized to:

```
["what is the name of"] ++
variants{["the current"];["this"]} ++ ["song"]
```

A good habit is to always include the system utterances whenever possible. This is done by opening the system grammar as a resource module and use the constants as macro definitions:

```
lin playlist_add = variants{
        reqVP playlist_add;
        req1x "add" (variants{[];["a song"] ++
                             variants{[];["to the playlist"]}})};
```

The term `playlist_add` following `reqVP` is the requested action from the system grammar, which is different from the rule that we are currently defining. The operation `reqVP` transforms a verb phrase from the resource grammar into a string, optionally adding "I want to. . . " and ". . . please". The operation `req1x` does the same on a verb phrase specified as two strings.

An alternative to using strings to specify utterances is to use syntax trees from the GF resource grammar:

```
lin playlist_add__song x = variants{
        reqVP (ComplV2 add_V2 x);
        reqVP (ComplV3 add_to_V3 x (the_N_sg playlist_N))};
```

An advantage with using only syntax trees is that they can be reused for different languages.

GoDiS semantics

We define the semantics for user utterances in terms of the semantics for system utterances. This means that we open the semantics system grammar as a resource module and give the semantics as syntax trees.

```
lin current_song = pm1 (ask current_song_Q);
    playlist_add = pm1 (request playlist_add);
    artist     x = pm1 (shortAns (artist x));
```

The operation `pm1` constructs a single-element list from one argument. There are also the operations `pm2`, `pm3`, etc. for constructing multiple-argument lists. The functions `ask` and `request` are defined in the common system grammar, and `current_song_Q` and `playlist_add` are defined in the MP3 system grammar.

Combined moves taking extra arguments are handled in the same way, only returning longer lists of dialogue moves:

```
lin song_artist x y = pm2 (shortAns (song x))
                          (shortAns (artist y));
    playlist_add__song_artist x y = pm3 (request playlist_add)
                                        (answer (song_to_add_P x))
                                        (answer (artist_to_add_P y));
    available_song__artist x = pm2 (ask available_song_Q)
                                   (shortAns (artist x));
```

4.5 Integrating multimodality in the grammars

The grammar library supports both flavours of multimodality, parallel and integrated as introduced in TALK deliverable D1.2a (Bringert et al., 2005). Parallel multimodality, such as graphical output, is handled as just another concrete syntax.

Integrated multimodality, such as spoken user input integrated with clicks on a screen, is handled by adding a row to the linearization record for the input clicks. This is already incorporated in the GF resource grammar, as explained in section 1.3.7.

Both these techniques are thoroughly discussed in TALK deliverable D1.2b (Ljunglöf et al., 2006), and will not be discussed further in this chapter.

4.6 Extending the GF grammar library

It is possible to extend the existing GF library with new languages or domains. By a new language we mean any one of the languages covered by the GF resource grammar. This is because the system grammars are specified using the resource grammar.

Adding a new language to the GF grammar library

We assume that the language to be added is Finnish. If the language is not already present in the grammar library, we have to add the following grammars to the `Common` directory:

- Concrete syntax for the resource grammar, `GodisLangFin.gf`
- Concrete syntax for the system grammar, `GodisSystemFin.gf`
- Concrete syntax for the user grammar, `GodisUserFin.gf`

The last two grammars are very straightforward to create, and require no knowledge about the language. The resource grammar however, requires enough knowledge about Finnish to translate phrases like "I thought you said" or the different possible variants of "I want to".

We now assume that the language already is in the library, but not in the domain in question, which we assume to be the MP3 domain. The following grammars have to be added to the `MP3` directory:

- A new lexicon syntax, `MP3LexiconFin.gf`
- A new syntax for the database resources, `MusicFin.gf`
- A new syntax for system utterances, `MP3SystemFin.gf`
- A new syntax for user utterances, `MP3UserFin.gf`

For the system utterances all you have to do is to add a grammar `MP3SystemFin` with the following content:

```
concrete MP3SystemFin of MP3System =
        GodisSystemFin, MusicFin **
        MP3SystemI with (Grammar=GrammarFin),
                        (GodisLang=GodisLangFin),
                        (MP3Lexicon=MP3LexiconFin);
```

This will give us a complete Finnish system grammar, provided there is a Finnish version of the common grammars `GodisLang` and `GodisSystem`, and the domain grammars `Music` and `MP3Lexicon`.

The most difficult part is to write the user grammar `MP3UserFin`. The simplest alternative is to only use syntax trees from the system grammar, but to get robust recognition we have to add the utterances and idioms which are common for Finnish MP3 dialogues.

Adding a new domain to the GF Grammar library

Adding a new domain consists in creating the GoDiS dialogue application and writing the GF grammars. We assume that the GoDiS application already exists and the problem only is to write the grammars. Assuming the example domain is a computerized psychotherapist which we can call *Psycho*, the following have to be created:

- A directory `Psycho`
- GF grammars for domain-specific lexicon entries:
  - Abstract syntax, `PsychoLexicon.gf`
  - One concrete syntax for each language: `PsychoLexiconEng.gf`, `PsychoLexiconSwe.gf`, ...
- GF grammars for system utterances:
  - Abstract syntax, `PsychoSystem.gf`
  - (Semantics, `PsychoSystemSem.gf`)
  - Language-independent syntax, `PsychoSystemI.gf`
  - (One concrete syntax for each language: `PsychoSystemEng.gf`, `PsychoSystemSwe.gf`, ...)
- GF grammars for user utterances:
  - Abstract syntax, `PsychoUser.gf`
  - (Semantics, `PsychoUserSem.gf`)
  - One concrete syntax for each language: `PsychoUserEng.gf`, `PsychoUserSwe.gf`, ...

The files in parenthesis are very straightforward to write – they can in fact be created automatically from the abstract syntax. The sizes of the files depend largely on the size of the domain – roughly the sizes of the grammars are proportional to the number of actions, predicates, sorts and individuals there are in the domain.

Chapter 5

Summary and Conclusions

The ISU approach uses abstract representations for dialogue states and update rules which allow the generic characterisation of flexible dialogue strategies. This enables the same code for dialogue management techniques to be used for different natural languages and for different domains.

In this deliverable we have discussed how dialogue systems, and especially grammars for dialogue systems, can be related to existing knowledge representation systems. We have focussed on one specific language for knowledge representation, the Web Ontology Language (OWL), which is a W3C standard for ontology descriptions for knowledge representation. Since it is a standard there is already much work done on relating OWL to other ontology formalisms.

We have shown how to generate the domain-specific utterances for a device-oriented dialogue system from an ontology describing the devices. An example system using this technique is Mimus, a dialogue system with which one can control different devices in a home, such as lights, alarms and washing machines. All information about devices is specified in an ontology, which can be updated on the fly with e.g. new devices or new information about existing devices.

We have also shown that all domain-specific utterances in a domain for the GoDiS dialogue system can be specified as a single ontology. By representing the utterances as abstract syntax trees in a Grammatical Framework (GF) resource grammar, the representation of utterances is language-independent. The resource grammar we are using has a coverage comparable to the Core Language Engine, and exists for 11 different languages, including the TALK languages English, German, Spanish, and Swedish, but also the non-Indoeuropean languages Finnish, Russian and Arabic. Since the grammar is multilingual, this means that the only thing that has to be done to localize a dialogue application to a new language is to write a lexicon for the domain-specific entities.

In this deliverable we have also described the final status of the full TALK Grammar Library, which is written in GF. The structure has been modified so that the domain-specific parts of a grammar can be described in an OWL ontology, and to further simplify localization to new languages, domains and modalities.

Bibliography

Amores, G. and Quesada, J. F. (1997). Episteme. *Proceedings of Procesamiento del Lenguaje Natural*, (21):1–16.

Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic Handbook*. Cambridge University Press.

Bernstein, A., Kaufmann, E., Kaiser, C., and Kiefer, C. (2006). Ginseng: A guided input natural language search engine for querying ontologies. In *Jena User Conference*, Bristol, UK.

Brickley, D. and Guha, R. V., editors (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, `http://www.w3.org/TR/rdf-schema/`.

Bringert, B., Cooper, R., Ljunglöf, P., and Ranta, A. (2005). Development of multimodal and multilingual grammars: viability and motivation. Deliverable D1.2a, TALK Project.

Coppi, S., Noia, T. D., Sciascio, E. D., Donini, F., and Pinto, A. (2005). Ontology-based natural language parser for e-marketplaces. In *18th Intl. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, volume 3533, pages 279–289. Springer-Verlag.

Dean, M. and Schreiber, G., editors (2004). *OWL Web Ontology Language Reference*. W3C Recommendation, `http://www.w3.org/TR/owl-ref/`.

Denecke, M. (2002). Rapid prototyping for spoken dialogue systems. In *19th International Conference on Computational Linguistics*, Taipei, Taiwan.

Dymetman, M., Lux, V., and Ranta, A. (2000). XML and multilingual document authoring: Convergent trends. In *COLING*, pages 243–249, Saarbrücken, Germany.

Estival, D., Nowak, C., and Zschorn, A. (2004). Towards ontology-based natural language processing. In *RDF/RDFS and OWL in Language Technology: 4th Workshop on NLP and XML*. ACL.

Johansson, M. (2006). Globalization and localization of a dialogue system using a resource grammar. Master's Thesis, Computational Linguistics, Gothenburg University.

Khegai, J., Nordström, B., and Ranta, A. (2003). Multilingual syntax editing in GF. In Gelbukh, A., editor, *CICLing–2003: Intelligent Text Processing and Computational Linguistics*, LNCS 2588, pages 453–464. Springer.

Ljunglöf, P., Amores, G., Cooper, R., Hjelm, D., Manchón, P., Pérez, G., and Ranta, A. (2006). Multimodal grammar library. Deliverable D1.2b, TALK Project.

Ljunglöf, P., Bringert, B., Cooper, R., Forslund, A.-C., Hjelm, D., Jonsson, R., Larsson, S., and Ranta, A. (2005). The TALK grammar library: an integration of GF with TrindiKit. Deliverable D1.1, TALK Project.

Milward, D., Amores, G., Blaylock, N., Larsson, S., Ljunglöf, P., Manchón, P., and Pérez, G. (2006). Dynamic multimodal interface reconfiguration. Deliverable D2.2, TALK Project.

Milward, D. and Beveridge, M. (2003). Ontology-based dialogue systems. In *IJCAI-03 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Acapulco, Mexico.

Milward, D. and Beverige, M. (2003). Ontology–based dialogue systems. In *IJCAI 2003 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*.

Nordström, B., Petersson, K., and Smith, J. (1990). *Programming in Martin-Löf's Type Theory*. Oxford University Press.

Patel-Schneider, P. F., Hayes, P., and Horrocks, I., editors (2004). *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, http://www.w3.org/TR/owl-semantics/.

Quesada, J. F. and Amores, G. (2002). Knowledge–based reference resolution for dialogue management in a home domain environment. In Johan Bos, M. E. and Matheson, C., editors, *Proceedings of the sixth workshop on the semantics and pragmatics of dialogue (Edilog)*, pages 145–189.

Ranta, A. (1994). *Type-Theoretical Grammar*. Oxford University Press.

Ranta, A. (2004). Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Rayner, M., Carter, D., Bouillon, P., Digalakis, V., and Wirén, M. (2000). *The Spoken Language Translator*. Cambridge University Press.

Russ, T., Valente, A., MacGregor, R., and Swartout, W. (1999). Practical experiences in trading off ontology usability and reusability. In *Proceedings of the Knowledge Acquisition Workshop (KAW99)*, Banff, Alberta.

Seki, H., Matsumara, T., Fujii, M., and Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Sharon Oviatt, S. L., D. A. and Kuhn, K. (1997). Integration and synchronization of input modes during multimodal human-computer interaction. In *Proceedings of Conference on Human Factors in Computing Systems: CHI '97*.

Williem Adriaans, P. (1992). *Language Learning from a Categorial Perspective*. PhD thesis, Amsterdam University.

Zaanen, M. V. (2000). Abl: Alignment-based learning. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, Saarbrücken.

Appendix A

The enhanced multimodal grammar library

A.1 Downloading the grammar library

The TALK Enhanced Multimodal Grammar Library can be downloaded from

`http://www.ling.gu.se/projekt/talk/software/`

The distribution consists of a collection of GF grammar modules, distributed in the following directories:

- Common grammars for GoDiS-based dialogue systems
- The application domains MP3 (for the *DJGoDiS* application), Agenda (for the *AgendaTalk* application), and Tram (for the *TramGoDiS* application)

The directories and grammar modules are described in more detail in chapter 4.

OWL ontology for a GoDiS dialogue domain

Apart from the GF grammars, the library also consists of an example OWL ontology for the *TramGoDiS* application. This ontology is located in the directory OWL, and can be opened by any OWL compliant ontology editor, such as Protegé.[1]
The structure of the ontology is described in more detail in section 3.4.

A.2 Installation instructions

First download and install Grammatical Framework. Source code, binaries and installation instructions can be found on the GF homepage:

[1] `http://protege.stanford.edu/`

http://www.cs.chalmers.se/~aarne/GF/

Set the search path to the GF library, and start GF from inside the directory of the grammar files:

- In csh, tcsh:

```
> setenv GF_LIB_PATH (path-to-GF)/lib
> (path-to-GF)/bin/gf
Welcome to Grammatical Framework, Version 2.4
...
```

- In bash:

```
$ export GF_LIB_PATH=(path-to-GF)/lib
$ (path-to-GF)/bin/gf
Welcome to Grammatical Framework, Version 2.4
...
```

If GF will be used on a regular basis, the `gf` binary should be added to the global search path and the environment variable `GF_LIB_PATH` should be set globally.

A.3 Testing the grammars

The grammars can be tested separately by loading them into GF. The relevant concrete syntax modules are:

> *DomSrcLng*.gf, where *Dom* ∈ {MP3, Agenda, Tram}, *Src* ∈ {System, User}, and *Lng* ∈ {Eng, Swe, Sem}.

The following is an example of the capabilities of the GF program. For more information about how to use GF, see the GF documentation. This example assumes we are testing the *DJGoDiS* grammar for user utterances, which of course can be replaced by any of the other grammars in the library.

1. Start GF in the directory where the grammars are located:

```
$ cd (path-to-grammar-library)/MP3/
$ gf
```

2. Load the source module(s) into GF:

```
> i MP3UserSem.gf
> i MP3UserEng.gf
> i MP3UserSwe.gf
```

3. Select the English concrete grammar:

```
> sf -lang=MP3UserEng
```

4. Parse an English utterance:

```
> p "play like a prayer by madonna"
request_S (play_item__song_artist like_a_prayer madonna)
```

5. Translate (i.e. parsing followed by linearization) from English to Swedish:

```
> p "play like a prayer by madonna" | l -all -lang=MP3UserSwe
spela like a prayer med madonna / ...
```

The option `-all` shows all possible variants of linearizing a syntax term.

6. Translate from English to GoDiS dialogue moves:

```
> p "play like a virgin by madonna" | l -lang=MP3UserSem
[request(play), answer(song(like_a_virgin)), answer(artist(madonna))]
```

In this case we don't need the option `-all`, since there is only one possible variant for the semantics.

7. Generate 5 random Swedish utterances:

```
> gr -number=5 | l -lang=MP3UserSwe
in the city med eagle eye cherry
rant radio
va
jag vill ändra balansen mitten tack
jag vill spela nummer tre tack
```

8. Quit GF:

```
> q
```