Multimodal Dialogue System Grammars*

Björn Bringert, **Peter Ljunglöf**, **Aarne Ranta**
Department of Computing Science
Chalmers University of Technology
and Göteborg University
{bringert,peb,aarne}@cs.chalmers.se

Robin Cooper
Department of Linguistics
Göteborg University
cooper@ling.gu.se

Abstract

We describe how multimodal grammars for dialogue systems can be written using the Grammatical Framework (GF) formalism. A proof-of-concept dialogue system constructed using these techniques is also presented. The software engineering problem of keeping grammars for different languages, modalities and systems (such as speech recognizers and parsers) in sync is reduced by the formal relationship between the abstract and concrete syntaxes, and by generating equivalent grammars from GF grammars.

1 Introduction

We are interested in building multilingual multimodal grammar-based dialogue systems which are clearly recognisable to users as the same system even if they use the system in different languages or in different domains using different mixes of modalities (e.g. in-house vs in-car, and within the in-house domain with vs without a screen for visual interaction and touch/click input). We wish to be able to guarantee that the functionality of the system is the same under the different conditions.

Our previous experience with building such multilingual dialogue systems is that there is a software engineering problem keeping the linguistic coverage in sync for different languages. This problem is compounded by the fact that for each language it is normally the case that a dialogue system requires more than one grammar, e.g. one grammar for speech recognition and another for interaction with the dialogue manager. Thus multilingual systems become very difficult to develop and maintain.

In this paper we will explain the nature of the Grammatical Framework (GF) and how it may provide us with a solution to this problem. The system is oriented towards the writing of multilingual and multimodal grammars and forces the grammar writer to keep a collection of grammars in sync. It does this by using computer science notions of abstract and concrete syntax. Essentially abstract syntax corresponds to the domain knowledge representation of the system and several concrete syntaxes characterising both natural language representations of the domain and representations in other modalities are related to a single abstract syntax.

GF has a type checker that forces concrete syntaxes to give complete coverage of

*This project is supported by the EU project TALK (Talk and Look, Tools for Ambient Linguistic Knowledge), IST-507802

the abstract syntax and thus will immediately tell the grammar writer if the grammars are not in sync. In addition the framework provides possibilities for converting from one grammar format to another and for combining grammars and extracting sub-grammars from larger grammars.

2 The Grammatical Framework and multilingual grammars

The main idea of Grammatical Framework (GF) is the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

As an example of a GF representation, the following abstract syntax tree represents a possible user input in our example dialogue system.

```
GoFromTo
  (PStop Chalmers)
  (PStop Valand)
```

The English concrete syntax relates the query to the string

```
I want to go from Chalmers
to Valand
```

The Swedish concrete syntax relates it to the string

```
Jag vill åka från Chalmers
till Valand
```

The strings are generated from the tree in a compositional rule-to-rule fashion. The generation rules are automatically inverted to parsing rules.

The abstract theory of Grammatical Framework (Ranta, 2004) is a version of dependent type theory, similar to LF (Harper et al., 1993), ALF (Magnusson and Nordström, 1994) and COQ (Coq, 1999). What GF adds to the logical framework is the possibility of defining concrete syntax. The expressiveness of the concrete syntax has developed into a functional programming language, similar to a restricted version of programming languages like Haskell (Peyton Jones, 2003) and ML (Milner et al., 1997).

The separation between abstract and concrete syntax was suggested for linguistics in (Curry, 1963), using the terms "tectogrammatical" and "phenogrammatical" structure. Since the distinction has not been systematically exploited in many well-known grammar formalisms, let us summarize its main advantages.

Higher-level language descriptions The grammar writer has a greater freedom in describing the syntax for a language. As illustrated in figure 1, when describing the abstract syntax he/she can choose not to take certain language specific details into account, such as inflection and word order. Abstracting away smaller details can make the grammars simpler, both to read and understand, and to create and maintain.

Multilingual grammar writing It is possible to define several different concrete syntax mappings for one particular abstract syntax. The abstract syntax could e.g. give a high-level description of a family of similar languages, and each concrete mapping gives a specific language instance, as shown in figure 2.

This kind of multilingual grammar can be used as a model for interlingual translation between languages. But we do not have to restrict ourselves to only multilingual grammars; different concrete syntaxes can be given for different modalities. As an example, consider a grammar for displaying time table information. We can have one concrete syntax for writing the information as plain text, but we could also present the information in the form of a table output as a LaTeX file or in Excel format, and a third possibility is to output the information in a format suitable for speech synthesis.

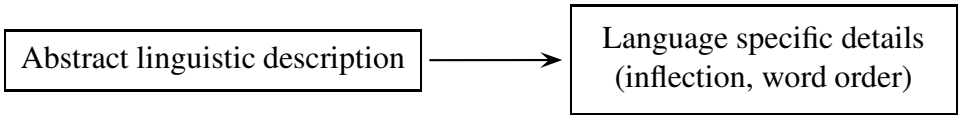

Figure 1: Higher-level language descriptions

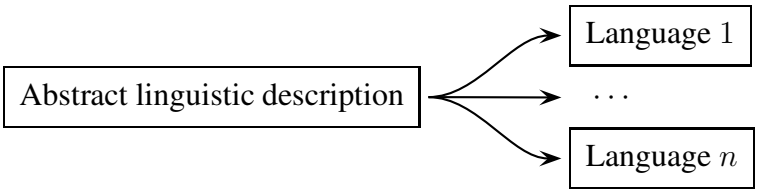

Figure 2: Multilingual grammars

Several descriptional levels Having only two descriptional levels is not a restriction; this can be generalized to as many levels as is wanted, by equating the concrete syntax of one grammar level with the abstract syntax of another level. As an example we could have a spoken dialogue system with a semantical, a syntactical, a morphological and a phonological level. As illustrated in figure 3, this system has to define three mappings; *i*) a mapping from semantical descriptions to syntax trees; *ii*) a mapping from syntax trees to sequences of lexical tokens; and *iii*) a mapping from lexical tokens to lists of phonemes.
This formulation makes grammars similar to transducers (Karttunen et al., 1996; Mohri, 1997) which are mostly used in morphological analysis, but have been generalized to dialogue systems by (Lager and Kronlid, 2004).

Grammar composition A multi-level grammar as described above can be viewed as a "black box", where the intermediate levels are unknown to the user. Then we are back in our first view as a grammar specifying an abstract and a concrete level together with a mapping. In this way we can talk about *grammar composition*, where the composition $G_2 \circ G_1$ of two grammars is possible if the abstract syntax of G_2 is equal to the concrete syntax of G_1.

If the grammar formalism supports this, a composition of several grammars can be precompiled into a compact and efficient grammar which doesn't have to mention the intermediate domains and structures. This is the case for e.g. finite state transducers, but also for GF as has been shown by (Ranta, 2005).

Resource grammars The possibility of separate compilation of grammar compositions opens up for writing *resource grammars* (Ranta, 2005). A resource grammar is a fairly complete linguistic description of a specific language. Many applications do not need the full power of a language, but instead want to use a more well-behaved subset, which is often called a *controlled language*. Now, if we already have a resource grammar, we do not even have to write a concrete syntax for the desired controlled language, but instead we can specify the language by mapping structures in the controlled language into structures in the resource grammar, as shown in figure 4.

3 Extending multilinguality to multimodality

Parallel multimodality *Parallel multimodality* is a straightforward instance of multilinguality. It means that the concrete syntaxes associated with an abstract syntax are not just different natural languages, but different representation modalities, encoded by language-like notations such as graphic

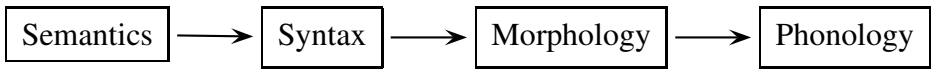

Figure 3: Several descriptional levels

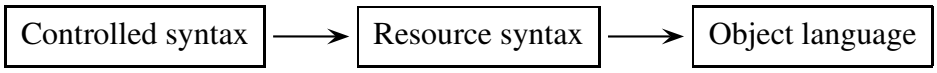

Figure 4: Using resource grammars

representation formalisms. An example of parallel multimodality is given below when a route is described, in parallel, by speech and by a line drawn on a map. Both descriptions convey the full information alone, without support from the other.

This raises the dialogue management issue of whether all information should be presented in all modalities. For example, in the implementation described below all stops are indicated on the graphical presentation of a route whereas in the natural language presentation only stops where the user must change are presented. Because GF permits the suppression of information in concrete syntax, this issue can be treated on the level of grammar instead of dialogue management.

Integrated multimodality *Integrated multimodality* means that one concrete syntax representation is a combination of modalities. For instance, the spoken utterance *"I want to go from here to here"* can be combined with two pointing gestures corresponding to the two *"here"*s. It is the two modalities in combination that convey the full information: the utterance alone or the clicks alone are not enough.

How to define integrated multimodality with a grammar is less obvious than parallel multimodality. In brief, different modality "channels" are stored in different fields of a record, and it is the combination of the different fields that is sent to the dialogue system parser.

4 Proof-of-concept implementation

We have implemented a multimodal route planning system for public transport networks. The example system uses the Göteborg tram/bus network, but it can easily be adapted to other networks. User input is handled by a grammar with integrated speech and map click modalities. The system uses a grammar with parallel speech and map drawing modalities. The user and system grammars are split up into a number of modules in order to simplify reuse and modification.

The system is also multilingual, and can be used in both English and Swedish. For every English concrete module shown below, there is a corresponding Swedish module. The system answers in the same language as the user made the query in.

In addition to the grammars shown below, the application consists of a number of agents which communicate using OAA (Martin et al., 1999). The grammars are used by the Embedded GF Interpreter (Bringert, 2005) to parse user input and generate system output.

4.1 Transport network

The transport network is represented by a set of modules which are used in both the query and answer grammars. Since the transport network is described in a separate set of modules, the Göteborg transport network may be replaced easily. We use **cat** judgements to declare categories in the abstract syntax.

```
abstract Transport = {
 cat
  Stop;
```

```
}
```

The Göteborg transport network grammar extends the generic grammar with constructors for the stops. Constructors for abstract syntax terms are declared using **fun** judgements.

```
abstract Gbg = Transport ** {
 fun
  Angered : Stop;
  AxelDahlstromsTorg : Stop;
  Bergsjon : Stop;
  ...
}
```

4.2 Multimodal input

User input is done with integrated speech and click modalities. The user may use speech only, or speech combined with clicks on the map. Clicks are expected when the user makes a query containing *"here"*.

Common declarations The `QueryBase` module contains declarations common to all input modalities. The `Query` category is used to represent the sequentialization of the multimodal input into a single value. The `Input` category contains the actual user queries, which will have multimodal representations. The `Click` category is also declared here, since it is used by both the click modality and the speech modality, as shown below.

```
abstract QueryBase = {
 cat
  Query;
  Input;
  Click;
 fun
  QInput : Input -> Query;
}
```

Since `QueryBase` is language neutral and common to the different modalities, it has a single concrete syntax. In a concrete module, **lincat** judgements are used to declare the linearization type of a category, i.e. the type of the concrete representations of values in the category. Note that different categories may have different linearization types. The concrete representation of abstract syntax terms is declared by **lin** judgements for each constructor in the abstract syntax.

Values in the `Input` category, which are intended to be multimodal, have records with one field per modality as their concrete representation. The `s1` field contains the speech input, and the `s2` field contains the click input. Terms constructed using the `QInput` constructor, that is sequentialized multimodal queries, are represented as the concatenation of the representations of the individual modalities, separated by a semicolon.

```
concrete QueryBaseCnc of QueryBase = {
 lincat
  Query = { s : Str };
  Input = { s1 : Str; s2 : Str };
  Click = { s : Str };
 lin
  QInput i = { s = i.s1 ++ ";" ++ i.s2 };
}
```

Click modality Click terms contain a list of stops that the click might refer to:

```
abstract Click = QueryBase ** {
 cat
  StopList;
 fun
  CStops : StopList -> Click;
  NoStop : StopList;
  OneStop : String -> StopList;
  ManyStops : String -> StopList -> StopList;
}
```

The same concrete syntax is used for clicks in all languages:

```
concrete ClickCnc of Click = QueryBaseCnc ** {
 lincat
  StopList = { s : Str };
 lin
  CStops xs = { s = "[" ++ xs.s ++ "]" };
  NoStop = { s = "" };
  OneStop x = { s = x.s };
  ManyStops x xs = { s = x.s ++ "," ++ xs.s };
}
```

Speech modality The `Query` module adds basic user queries and a way to use a click to indicate a place.

```
abstract Query = QueryBase ** {
 cat
  Place;
 fun
  GoFromTo : Place -> Place -> Input;
  GoToFrom : Place -> Place -> Input;
  PClick   : Click -> Place;
}
```

This module has a concrete syntax using English speech. Like terms in the `Query` category, `Place` terms are linearized to records with two fields, one for each modality.

```
concrete QueryEng of Query = QueryBaseCnc ** {
 lincat
  Place = {s1 : Str; s2 : Str};
 lin
  GoFromTo x y = {
   s1 = ["i want to go from"] ++ x.s1
        ++ "to" ++ y.s1;
   s2 = x.s2 ++ y.s2
  };
  GoToFrom x y = {
   s1 = ["i want to go to"] ++ x.s1
        ++ "from" ++ y.s1;
   s2 = x.s2 ++ y.s2
  };
  PClick c = { s1 = "here"; s2 = c.s };
}
```

Indexicality To refer to her current location, the user can use *"here"* without a click, or omit either origin or destination. The system is assumed to know where the user is located. Since *"here"* may be used with or without a click, inputs with two occurrences of *"here"* and only one click are ambiguous. A query might also be ambiguous even if it can be parsed unambiguously, since one click can correspond to multiple stops when the stops are close to each other on the map.

These are the abstract syntax declarations for this feature (in the `Query` module):

```
fun
 PHere    : Place;
 ComeFrom : Place -> Input;
 GoTo     : Place -> Input;
```

The English concrete syntax for this is added to the `QueryEng` module. Note that the click (`s2`) field of the linearization of an indexical *"here"* is empty, since there is no click.

```
lin
 PHere = { s1 = "here" ; s2 = [] };
 ComeFrom x = {
  s1 = ["i want to come from"] ++ x.s1;
  s2 = x.s2
 };
 GoTo x = {
  s1 = ["i want to go to"] ++ x.s1;
  s2 = x.s2
 };
```

Tying it all together The `TransportQuery` module ties together the transport network, speech modality and click modality modules.

```
abstract TransportQuery
       = Transport, Query, Click ** {
 fun
  PStop : Stop -> Place;
}
```

4.3 Multimodal output

The system's answers to the user's queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent. The information presented in the two modalities is however not identical, as the spoken output only contains information about where to change trams/buses. The map output shows the entire path, including intermediate stops.

Abstract syntax for routes The abstract syntax for answers (routes) contains the information needed by all the concrete syntaxes. All concrete syntaxes might not use all of the information. A route is a non-empty list of legs, and a leg consists of a line and a list of at least two stops.

```
abstract Route = Transport ** {
 cat
  Route;
  Leg;
  Line;
  Stops;
 fun
  Then : Leg -> Route -> Route;
  OneLeg : Leg -> Route;
  LineLeg : Line -> Stops -> Leg;
  NamedLine : String -> Line;
  ConsStop : Stop -> Stops -> Stops;
  TwoStops : Stop -> Stop -> Stops;
}
```

Concrete syntax for drawing routes The map drawing language contains sequences of labeled edges to be drawn on the map. The string *"drawEdge (6, [Chalmers, Vasaplatsen]); drawEdge (2, [Vasaplatsen, Gronsakstorget, Brunnsparken]);"* is an example of a string in the map drawing language described by the `RouteMap` concrete syntax.

The TransportLabels module extended by this module is a simple concrete syntax for stops.

```
concrete RouteMap of Route
       = TransportLabels ** {
 lincat
  Route, Leg, Line, Stops = { s : Str } ;
 lin
  Then l r = { s = l.s ++ ";" ++ r.s };
  OneLeg l = { s = l.s ++ ";" };
  LineLeg l ss =
   { s = "drawEdge" ++ "(" ++ l.s ++ ","
         ++ "[" ++ ss.s ++ "]" ++ ")" };
  NamedLine n = { s = n.s };
  ConsStop s ss = { s = s.s ++ "," ++ ss.s };
  TwoStops x y = { s = x.s ++ "," ++ y.s };
}
```

English concrete syntax for routes In the English concrete syntax we wish to list only the first and last stops of each leg of the route. The TransportNames module gives English representations of the stop names by replacing all non-English letters with the corresponding English ones in order to give the speech recognizer a fair chance.

```
concrete RouteEng of Route
       = TransportNames ** {
 lincat
  Route, Leg, Line = { s : Str } ;
  Stops = { start : Str; end : Str };
 lin
  Then l r = { s = l.s ++ "." ++ r.s } ;
  OneLeg l = { s = l.s ++ "." };
  LineLeg l ss =
   { s = "Take" ++ l.s ++ "from" ++ ss.start
         ++ "to" ++ ss.end };
  NamedLine n = { s = n.s };
  ConsStop s ss = { start = s.s;
                    end = ss.end };
  TwoStops s1 s2 = { start = s1.s;
                     end = s2.s };
}
```

5 Related Work

Johnston (1998) describes an approach to multimodal parsing where chart parsing is extended to multiple dimensions and unification is used to integrate information from different modalities. The approach described in this paper achieves a similar result by using records along with the existing unification mechanism for resolving discontinuous constituents. The main advantages of our approach are that it supports both parsing and generation, and that it does not require extending the existing formalism.

6 Conclusion

GF provides a solution to the problems named in the introduction to this paper. Abstract syntax can be used to characterise the linguistic functionality of a system in an abstract language and modality independent way. The system forces the programmer to define concrete syntaxes which completely cover the abstract syntax. In this way, the system forces the programmer to keep all the concrete syntaxes in sync. In addition, since GF is oriented towards creating grammars from other grammars, our philosophy is that it should not be necessary for a grammar writer to have to create by hand any equivalent grammars in different formats. For example, if the grammar for the speech recogniser is to be the same as that used for interaction with dialogue management but the grammars are needed in different formats, then there should be a compiler which takes the grammar from one format to the other. Thus, for example, we have a compiler which converts a GF grammar to Nuance's format for speech recognition grammars. The idea of generating context-free speech recognition grammars from grammars in a higher-level formalism has been described by Dowding et al. (2001), and implemented in the Regulus system (Rayner et al., 2003).

Another reason for using GF grammars has to do with the use of resource grammars and cascades of levels of representation as described in section 2. This allows for the hiding of grammatical detail from language and the precise implementation of modal interaction for other modalities. This enables the dialogue system developer to reuse previous grammar or modal interaction implementations without herself having to reprogram the details for each new dialogue system. Thus

the dialogue engineer need not be a grammar engineer or an expert in multimodal interfaces.

References

Björn Bringert. 2005. Embedded grammars. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, February.

The Coq Development Team, 1999. *The Coq Proof Assistant Reference Manual*. Available at `http://pauillac.inria.fr/coq/`

Haskell B. Curry. 1963. Some logical aspects of grammatical structure. In Roman Jacobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the 12th Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

John Dowding, Beth Ann Hockey, Jean Mark Gawron, and Christopher Culy. 2001. Practical issues in compiling typed unification grammars for speech recognition. In *Meeting of the Association for Computational Linguistics*, pages 164–171.

R. Harper, F. Honsell, and G. Plotkin. 1993. A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

Michael Johnston. 1998. Unification-based multimodal parsing. In *Proceedings of the 36th conference on Association for Computational Linguistics*, pages 624–630. Association for Computational Linguistics.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Torbjörn Lager and Fredrik Kronlid. 2004. The Current platform: Building conversational agents in Oz. In *2nd International Mozart/Oz Conference*, October.

Lena Magnusson and Bengt Nordström. 1994. The ALF proof editor and its proof engine. In *Types for Proofs and Program*, volume 806 of *LNCS*, pages 213–237. Springer.

David L. Martin, Adam J. Cheyer, and Douglas B. Moran. 1999. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, January–March.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML – Revised*. MIT Press, Cambridge, MA.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries*. Cambridge University Press, New York.

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

A. Ranta. 2005. Modular Grammar Engineering in GF. *Research in Language and Computation*. To appear.

Manny Rayner, Beth Ann Hockey, and John Dowding. 2003. An open-source environment for compiling typed unification grammars into speech recognisers. In *EACL*, pages 223–226.