UNIVERSITY OF GOTHENBURG

Adaptable Controlled Natural Languages for Online Query Systems

Master of Science Thesis in the Programme Computer Science

FAEGHEH HASIBI

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, October 2012.

Adaptable Controlled Natural Languages for Online Query Systems

FAEGHEH HASIBI

Examiner: AARNE RANTA

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2012

Abstract

This work introduces a technique for adapting GF-based query systems. This technique is implemented in a travel planning query system in two ways. Firstly, users can customize the system to their own needs and define synonyms for a series of information to use in later utterances. Secondly, the system can update GF grammar to make the system flexible when new situations occur. While implementing GF grammar adaptation in a query system, we introduce a design pattern for multilingual travel planning systems that allow users to find up-to-date travel plans. The resulting system can be easily ported to a new public transport network and communicates with a transport web service to find accurate travel plans. Adapting GF-based dialogue systems improves the functionality of speech recognizers by defining synonyms for specific phrases.

Contents

Acknowledgments

It is a pleasure to thank those who made this thesis possible. My special thanks go to my supervisor, Professor Aarne Ranta for his continual support and encouragements. His valuable comments and positive energy despite of his busy schedule was always the greatest assistance for me.

I would like to kindly appreciate Ramona Enache, who first introduced me to the language technology group. I want to specially thank Grégoire Détrez, Krasimir Angelov and John Camilleri for their kind behavior and undeniable help during my thesis. I am also grateful to Peter Ljunglöf for interesting discussions and for his ideas on dialog systems.

Last but not least, I want to thank my beloved family and specially my wonderful parents for all their continuous and vital supports. Even though I am far from them, their positive energy and encouragements are always with me.

Chapter 1

Introduction

A Controlled Natural Language (CNL) is a subset of a natural language which is designed to reduce the complexity and ambiguity of a full natural language and to include certain grammar rules and vocabulary terms [1] [2]. Controlled languages are designed to be used on specific domain, such as clinical practice [3], topography [4], touristic phrases [5] and public transportation queries [6]. The Gothenburg Tram Information System (GOTTIS) [6] is a multilingual multimodal dialogue system designed for public transport queries. This CNL application is based on Grammatical Framework (GF) [7], which is a grammar formalism used for multilingual grammars of controlled languages.

GOTTIS is an experimental application which uses GF grammars for interpreting user's input in different modalities (such as speech or map click). It uses a weighted directed graph for finding the shortest path through a subset of the Gothenburg public transportation network [8] [9]. Since this system does not support the complete transport network and departure times, it cannot be used for real travel planning. Moreover, the system is not flexible enough to support changes of a public transit system, such as routes, schedule and new vehicles. In order to have an adaptable dialogue system that presents up-to-date travel information, the GF grammars need to be updated automatically and during system execution.

The notion of adaptation is an important issue not only in transport dialogue systems but also in other dialogue systems, which must be updated. Moreover, most users need to adapt the dialogue system to their needs and communicate with the system by specific utterances. The idea of extending dialogue systems by allowing users to reconfigure the system to their interest is represented by voice programming [10]. This kind of adaptation is simply done when users define their own commands in speech dialogues. All in all, adaptation is highly demanded for controlled natural languages and must be managed in GF-based systems by adapting GF grammars. Accordingly, we introduce a technique for adapting GF-based Question Answering (QA) system, which is implemented in a QA transportation system.

The demonstrative system handles user adaptation with respect to the voice programming to support shorter and easier dialogues. In other words, it allows users to define a synonym for a series of information that may be used frequently in the dialogues. The synonyms will be saved by the system and can be used in later dialogues. The following conversations show a normal interaction with the system before applying user adaptation.

— U: I want to go from Chalmers to Valand today at 11:30

— S: Take tram number 7 from Chalmers track A to Valand track A at 11:31

However, these examples show how a user can record some commands and use them in later queries to have a laconic conversation. Meanwhile, the Swedish utterances depict multilingual aspect of the designed system.

— U: work means Chalmers on Monday at 7:30

— U: home means Valand

— U: Jag vill åka från hem till jobbet
(I want to go from home to work)

— S: Ta spårvagn nummer 10 från Valand läge B till Chalmers kl 07:33
(Take tram number 10 from Valand track B to Chalmers at 07:33)

According to voice programming definitions, a macro is “a way for the user to automate a complex task that he/she performs repeatedly or on a regular basis.” [10]. A list of supported macros by our travel planning system is stated below:

- Work means Chalmers.
  — (`VALUE` **means** `STOP-NAME`)

- Work means Chalmers on Monday.
  — (`VALUE` **means** `STOP-NAME DAY`)

- Work means Chalmers on Monday at 11:30.
  — (`VALUE` **means** `STOP-NAME DAY TIME`)

- Birthday means Saturday
  — (`VALUE` **means** `DAY`)

Our adaptation technique supports both user and system adaptation. Regarding system adaptation, the system can update GF grammars to adapt the system while facing unpredictable situations. For instance, new vehicle labels might be added to the transport network, which are not supported by the GF grammar. Encountering this situation, the system will adapt itself by adding the new vehicle label to the grammar.

In addition to adaptation, the resulting query system offers a design pattern for transport dialogue systems. This system communicates with transport web services to provide users an accurate travel plan. It is also completely portable to a new transport network by automatic generation of corresponding GF modules for presenting bus/tram stops. To perform this grammar generation, a list of bus/tram stops is quested from transport web service and then a GF writer application will write the information in a set of GF modules. Automatic generation of stop grammar eliminates the work of writing huge grammars by GF programmers when porting the system to a new transport network.

1.1 Grammatical Framework

This work is highly dependent on Grammatical Framework [7] for component generation. This section gives a short description of GF and its features.

Grammatical Framework (GF) is a type-theoretic grammar formalism based on Martin-Löf's theory [11]. GF can describe both formal and natural languages and is well suited for writing grammars of natural languages. However, the main strength of GF is multi-linguality that provides translation between several languages at the same time. It can also generate other necessary grammars like context-free grammars. GF grammars can be used for both parsing and generation of a language.

GF is a functional programming language similar to restricted version of ML [12] and Haskell [13]. It also gets some features of Java and C++. From programming language point of view, GF grammars can be used efficiently both in development and at run-time. Due to the incremental parsing algorithm [14] of grammars, language processing is polynomial in GF.

The key feature of GF is the distinction between two components of grammars: abstract syntax and concrete syntax. Every GF grammar has an ab-

stract syntax with one or more concrete syntaxes. The abstract syntax represents the semantic structure of the language, while the concrete syntax describes the specific features of a language.

One of the biggest achievements of GF is the resource grammar library [15], which provides the main grammar rules for a wide variety of natural languages. Currently, the library covers the morphology and syntax of 25 languages. The purpose of this library is to enable writing grammars without knowing the linguistic aspect of a particular language.

The GF software system can generate several file formats, such as `.gfo` and `.pgf`. The first one is GF object files, which are generated by importing source files, suffixed `.gf`. These object files are faster to load in comparison to source files. Additionally, GF grammars can be compiled to the Portable Grammar Format (PGF) [16] which is a low-level binary format, suffixed `.pgf`.

The GF package consists of three components: the compiler, the command interpreter and the run-time system. The compiler translates GF source files to object files, run time grammars (`.pgf`) and other formats. The command interpreter, namely the GF shell, provides a set of commands for parsing, linearization, visualization of parse tree, word alignments, etc. On the other hand, the run-time system performs parsing, translation and other functions with PGF grammars [7].

A PGF file is generated by compiling a set of concrete grammars that have the same abstract syntax. Consequently, it can be loaded and applied faster than bunch of GF grammar modules. In order to work with PGF files in other platforms, like mobile phones, a PGF interpreter is needed. A PGF interpreter can perform a subset of GF system functionality, such as parsing, linearization, random generation and type checking. PGF interpreters exist in Haskell, C, Java and JavaScript. Since Java is the host language for our system, we use JPGF library[1] to work with PGF embedded grammars.

[1] https://github.com/GrammaticalFramework/JPGF

1.2 Outline

The report is written in 7 chapters with the following descriptions:

Chapter 1: This chapter provides an introduction to the thesis. It also describes the nature of Grammatical Framework.

Chapter 2: In this chapter, we introduce a multi-lingual question answering system for planning journeys. It provides a profound description of a design pattern for travel planning dialogue systems as well as introducing a baseline system for applying adaptation in chapter 4. Moreover, it gives an introduction to the GF writer application, while the detailed information of this application is shown in appendix A.

Chapter 3: The idea of GF grammar adaptation is introduced in this chapter. Additionally, the patterns for adding or updating grammar rules are mentioned. However, the detail implementations are shown with some examples in chapter 4.

Chapter 4: Moving on chapter 2 and 3, we show a real example of GF grammar adaption in the online query system described in chapter 2. To illustrate the importance of grammar adaption, two usage of adaption are demonstrated: user adaptation and system adaptation.

Chapter 5, 6 and 7: The evaluation, conclusion and some suggestions future works of this thesis are illustrated in these chapters, respectively.

Chapter 2

The Baseline System

2.1 Introduction

We developed a multi-lingual Question Answering (QA) system which presents up-to-date travel plan. This system integrates GF grammars, a public transport service, speech synthesizer and an embeddable GF writer application. The GF writer is a part of this system that generates and modifies GF modules.

The purpose of developing this system is to support our adaptation techniques by experimenting on a GF based online query system. In addition, the system offers a design pattern for travel planning dialogue systems that can be easily adapted to new transport networks and natural languages.

The transport question answering system and the GF Writer application are described in this chapter. More information about GF writer methods is provided in appendix A.

2.2 System Overview

The main task in a question answering system is to extract required information from user's queries and construct an answer by querying a database. In our system, a natural language query must be converted to an acceptable format for a transport web service. Since each bus stop is identified by a unique number, stop names must be presented by their identifiers in the HTTP request. As a consequence, a mapping between bus stop names and identifiers is required. We perform both information extraction and bus stop mapping by translating a natural language query to a HTTP request using GF grammars.

The demonstration system introduces a pattern for connecting to a web service in a multilingual dialogue system. As it is shown in figure 1, the embedded PGF interpreter translates a user input to a HTTP request, which can be sent to the transport web service. Then travel information is retrieved from

the web service response. Putting all this information together, a parse tree is generated that can be linearized to a natural language answer and finally the output is fed to a speech synthesizer.

The PGF interpreter, shown below, is an embedded lightweight interpreter which supports parsing, linearization and translation.

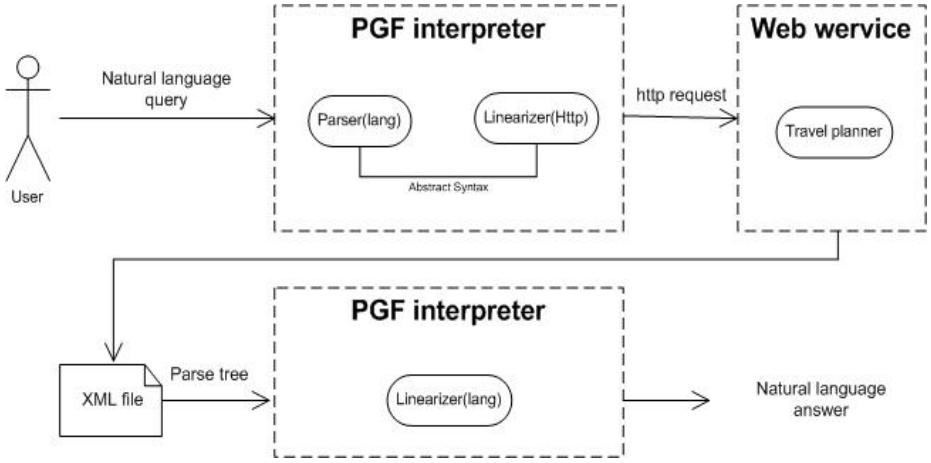

Fig. 1. Architecture overview of multilingual travel planning dialogue system

As a multi-lingual system, the system responses should be presented in different languages. Accordingly, system answers are constructed by linearizing parse trees to target languages. So, to port the system to a new language, all that is needed is defining a new concrete syntax. Since the HTTP language and natural languages have the same abstract syntax, both parsing and linearization can be done for phrases for a new language.

The described system uses the Gothenburg transport web service (vasttrafik.se) and supports queries in both English and Swedish. We use eSpeak [2]as a speech synthesizer to represent system response to the user.

In order to use this system for a new transport network, the bus/tram stops must be changed to new ones. To address this issue, the GF modules that

[2] http://espeak.sourceforge.net/

hold bus/tram stops are generated automatically by the GF writer application. The following chapter introduces the GF writer, with a focus on its architecture.

2.3 The GF Writer Application

The embeddable GF Writer is designed to dynamically construct and edit GF modules in a Java program. In other words, it can produce or update GF grammars during execution of a program. In order to generate a new module, essential parts of the grammar such as module header and body, flags, categories and function declarations should be defined. Similarly, for updating a GF grammar, module name and new function definition are needed. After generating or modifying GF grammar, a newly generated PGF file will be replaced with previous one.

The GF Writer can be used effectively in programs that use GF grammars. For instance, it can be used to apply changes to the GF grammar of spoken language translators, dialogue systems and localization purpose applications.

2.3.1 GF Writer Structure

The GF Writer offers three main classes for creation and modification of GF modules: *Abstract*, *Concrete* and *Customizer* class. *Abstract* and *Concrete* classes are designed for creation of abstract and concrete modules. In contrast, the *Customizer* class is aimed to update both abstract and concrete modules. Using methods of the *Customizer* class, grammars will be updated according to the user's requests.

We split up grammar rules into right and left hand side parts, which are named *RHS* and *LHS* in the *Element* class. Figure 2 illustrates more detailed information about part of the GF writer classes and their relations.

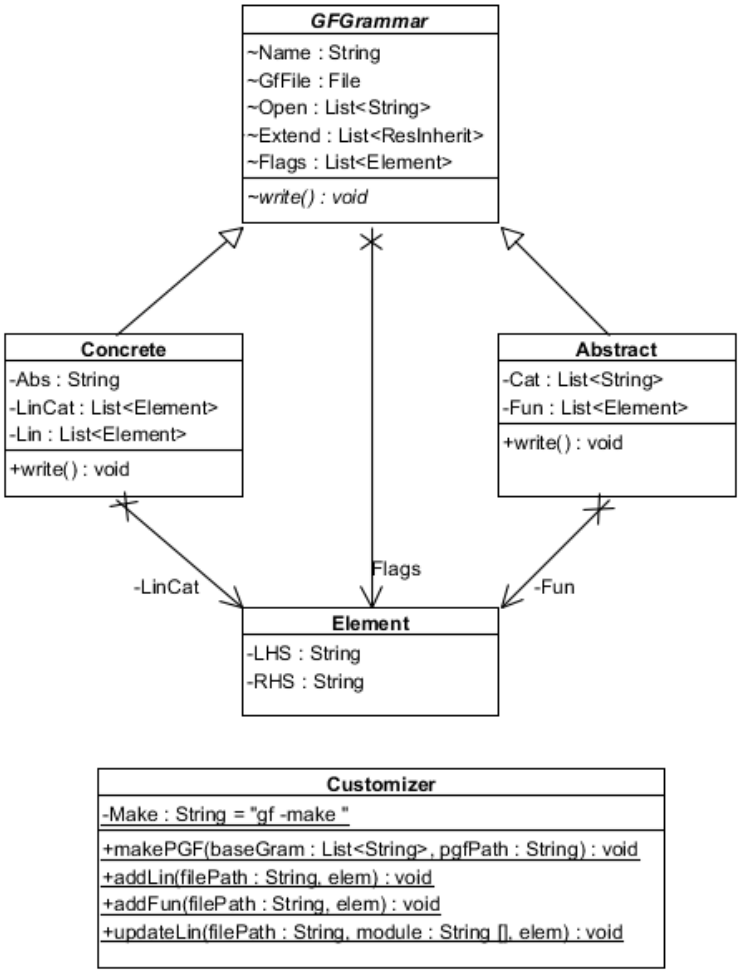

Fig. 2. A part of class diagram for GF Writer

The *GFGrammar* class has some properties to hold general information about a GF module, regardless of its type (abstract or concrete); that are module name and address, extension and opening modules and list of flags. These properties will be initialized when an *Abstract* or C*oncrete* instance is created.

As it is shown in figure 2, a list of *ResInherit objects* is used to present an extension module. The *ResInherit* class is designed to support restricted inher-

itance; that is, allowing a module to inherit a selection of function names. The GF system supports two types of restricted inheritance: include and exclude. In order to make a distinction between these types of inheritance, *InheritType* enumeration is designed. Figure 3 represents the structure of mentioned class and enumeration.

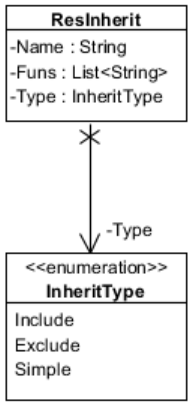

Fig. 3. Class diagram for Inheritance in GF Writer

More detailed information about GF Writer methods with some examples are documented in Appendix A.

2.4 Stop Grammar Generation

The embeddable GF writer application generates a set of abstract and concrete GF modules. For our transport dialogue system this application is used for creating a set of modules for describing the transport network. To reach this goal, the following steps are performed:

1. Sending a query to transport web service to get a list of all bus/tram stops available in the journey planner

2. Parsing XML file and retrieving bus/tram stops information

3. Generating transport network modules for both natural languages and HTTP request

Three classes are implemented to perform mentioned computations. These classes are placed in *StopGenerator* package and are visualized in the class

diagram shown in figure 4. The *static* fields and methods are marked by underlines in this representation of classes.

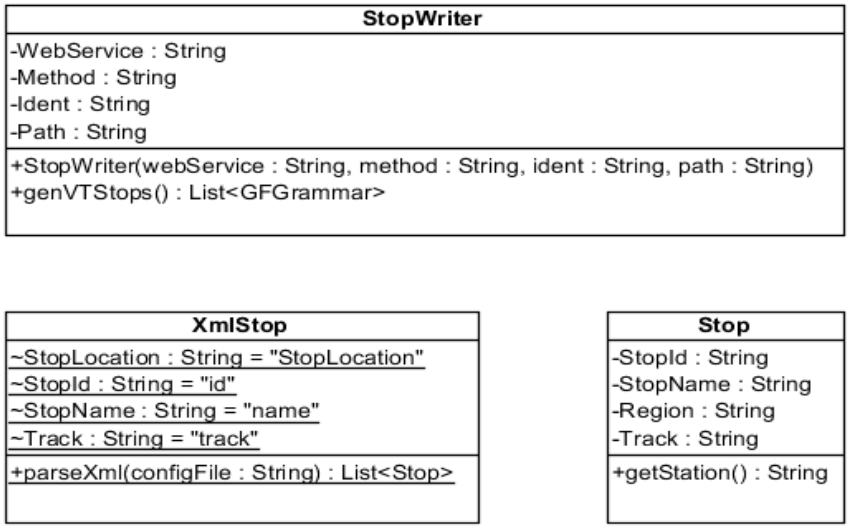

Fig. 4. Stop Generator class diagram

2.5 System Grammar

The transport system contains a set of modules for both query and answer grammars. The list of stops is described in a separate module that can be reproduced by GF writer application when porting system to a new transport web service. Due to the large number of stops, *Stop* modules are automatically generated by an embedded GF Writer. In the following subsections, the grammar used for this online query system is demonstrated.

2.5.1 Grammar overview

The query and answer grammars are divided into several modules to provide extendibility for developing more sophisticated systems. Figure 5 depicts an overview of grammar modules which has been produced with module dependency visualization feature in GF. Both Query and Answer grammars use the same modules for travel time and stop names. On top of these gram-

mars, the*Travel* grammar extends both *Query* and *Answer* grammars to put all grammar rules together.

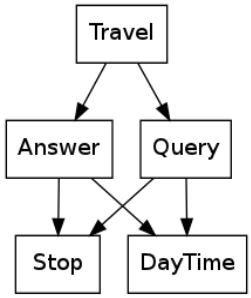

Fig. 5. Grammar design pattern for transport query system

The following figures show the details of grammar modules for *Query* and *Answer* grammars. In these figures, abstract and concrete syntaxes are marked by rectangles and ellipses, respectively. The dashed ellipses represent resource grammar modules, which package operation definitions as a reusable resource.

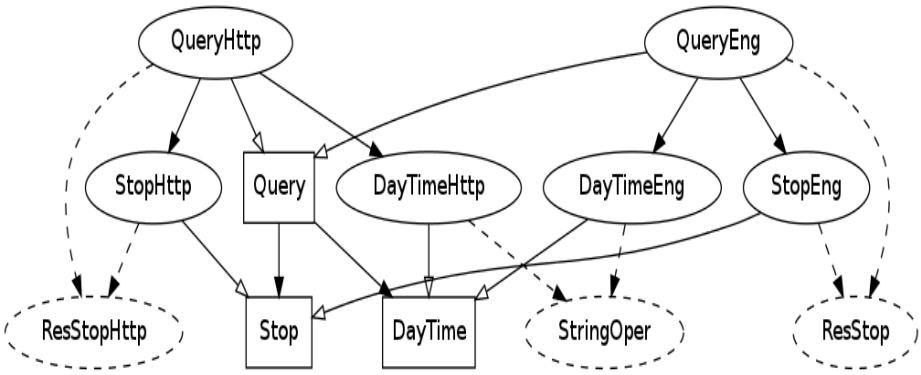

Fig. 6. Query Grammar modules for English and HTTP format

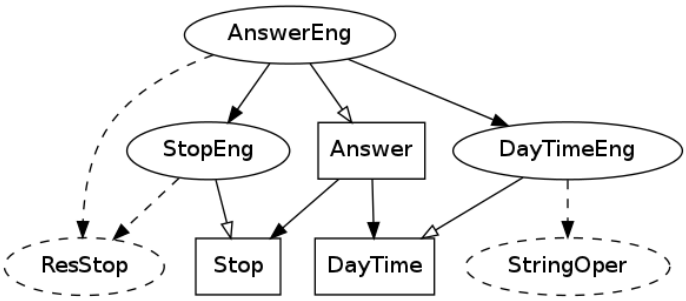

Fig. 7. Answer grammar modules for English language

Since system responses are generated from linearization of a parse tree, a seperate module for HTTP language (*AnswerHttp*) is not needed, which is shown in figure 7. However, the *QueryHttp* module is required for translating natural language queries to HTTP format.

In order to support Swedish dialogues, the Swedish concrete syntax is considered for the abstract syntax. Due to similarity of English and Swedish syntaxes, Swedish modules are not shown.

The details of grammar module are demonstrated in the following sections.

2.5.2 Stop Grammar

Each public transport system has a list of stops that must be presented in GF grammar. We represent the transport network in a separate set of modules to make the system portable to other transport systems. In contrast with previous works, the automatically generated abstract syntax offers a unique numerical function name for each stop. These numerical function names prevent ambiguity of stops with the same name which are different in other details. The *Stop* module holds the declarations of all bus/tram stops.

```
abstract Stop = {
cat
  Stop;
fun
  St_0 : Stop;
  St_1 : Stop;
  . . .
}
```

The English concrete module linearizes *Stop* terms in records with some objects. These objects present name, region and track of each stop. Using this structure, the query functions are free to use stop details (region and track number) or not. Stops with empty track name are used to parse utterances regardless of track name.

```
concrete StopEng of Stop = open ResStop in {
flags
  coding = utf8;
lincat
  Stop = ResStop.TStop;
lin
  St_0 = mkStop "Chalmers" "Göteborg" "B";
  St_1 = mkStop "Vettnet" "Strömstad" "track A";
  St_2 = mkStop "Vettnet" "Strömstad" "";
  . . .
}

resource ResStop = {
oper
  TStop = { s : Str; r : Str; t : Str; alt : Str};
  mkStop : Str -> Str -> Str -> TStop =
    \stop, region, track ->
       {s = stop; r = region; t = track};
}
```

The *StopHttp* concrete syntax contains stop identifiers. Due to the same abstract syntax for English and HTTP concrete syntaxes, each stop is simply mapped to its identifier.

```
concrete StopHttp of Stop = open ResStopHttp in {
flags
  coding = utf8;
lincat
  Stop = ResStopHttp.TStop;
lin
  St_0 = mkStop "9022014004420003";
  St_1 = mkStop "9021014004420000";
  . . .
}

resource ResStopHttp = {
oper
```

```
  TStop = { s : Str} ;
  mkStop : Str -> { s : Str} = \st -> { s = st };
}
```

The abstract and concrete modules of the stops grammar are generated automatically in the dialogue manager using GF Writer.

2.5.3 Date Time Grammar

To get a precise travel plan, the user should mention both the day and the time of the travel. Commonly, the user mentions week days or some adverbs such as today and tomorrow to refer to the date of the travel in a dialogue system. Regarding this fact, we encode each day to a number in the *DayTimeHttp* module and replace it in the HTTP request after calculating the corresponding date in our java program. This is the grammar that relates to the day and time of travel.

```
abstract DayTime = {
cat
  Day ;
  Time ;
  Number ;
  Digit ;
fun
  HourMin : Number -> Number -> Time; -- hour : min
  Hour : Number -> Time ;             -- 7 o'clock
  Today, Tomorrow : Day ;
  Saturday, Sunday, . . . , Friday : Day ;

  Num : Digit -> Number ;
  Nums : Digit -> Number -> Number;
  N0, N1, N2, . . . , N9 : Digit;
}
```

The *DayTimeHttp* module addresses how terms in the `Day` category are linearized to special numbers.

```
concrete DayTimeHttp of DayTime =  open StringOper
in {
lincat
```

```
  Digit, Number = { s : Str} ;
  Day = TDay ;
  Time = TTime ;
oper
  TDay = { s : Str};
  TTime = { s : Str};

lin
  HourMin hour min = {s = hour.s ++ ":" ++ min.s } ;
  Hour hour = { s= hour.s ++ "- 0 0" } ;
  Today     = ss "8" ;
  Tomorrow  = ss "9" ;

  Saturday  = ss "7" ;
  Sunday    = ss "1" ;
  . . .
  Friday    = ss "6" ;

  Num d = d ;
  Nums d n = {s = d.s ++ n.s};
  N0 = ss "0";
  . . .
  N9 = ss "9";
}
```

The English concrete module is also shown here. Since speech synthesizer can produce speech from numeral text, digits are similar to HTTP module digits and all are linearized to numbers. Due to this similarity, the functions are not shown in these lines of the code.

```
concrete DayTimeEng of DayTime {
lincat
  Digit, Number= { s : Str };
  Time = TTime;
  Day = TDay;
oper
  TDay = { s : Str; prep : Str};
  TTime = { s : Str };
lin
  HourMin hour min = {s = hour.s ++ ":" ++ min.s} ;
  Hour hour = {s = hour.s ++ "o'clock"};

  Today     = mkDay "today" "" ;
```

```
  Tomorrow  = mkDay "tomorrow" "" ;

  Saturday  = mkDay "Saturday" "on" ;
  . . .

oper
  mkDay : Str -> Str -> {s : Str; prep : Str} =
   \d,p -> {s = d; prep = p};
}
```

2.5.4 Query Grammar

The *QueryHttp* module is a part of our approach that generates HTTP request in collaboration with other concrete modules. The developed dialogue system uses Göteborg transport service as a travel finder. Since this web service supports HTTP GET method, the linearization rules in QueryHttp are targeted toward producing a GET request with the user's parameters. In this set of grammar, a function (`GoFromTo`) for producing a query and its linearization are shown.

```
abstract Query = DayTime, Stop ** {
flags
  startcat = Query;
cat
   Query ;
fun
  GoFromTo : Stop -> Stop -> Day -> Time -> Query ;
}
```

In the following concrete module, it is shown how a GET request is generated. The pattern for this request is defined according to the transport web service. However, for adding more supported queries of the web service to the system, all that needs to be done is to add corresponding functions to the English concrete module.

```
concrete QueryHttp of Query = DayTimeHttp, StopHttp
** open (R=ResStopHttp) in {
lincat
  Query = { s : Str} ;
lin
  GoFromTo = mkHttp Dep ;
```

```
oper

  mkHttp : SearchTyp -> R.TStop -> R.TStop
           -> TDay -> TTime -> { s : Str} =
    \ searchTyp, from, to, day, time ->
       {s = "date=" ++ day.s ++
            "&time=" ++ time.s ++
           "&originId=" ++ from.s ++
            "&destId=" ++ to.s };
}
```

The `GoFromTo` function is linearized as bellow in the *QueryEng* module.

```
GoFromTo from to day time =
  {s = "I want to go from" ++ from.s ++ from.r ++
       "to" ++ to.s ++ to.r ++
       day.prep ++ day.s ++
       "at" ++ time.s };
```

2.5.5 Answer Grammar

System response utterances are implemented in *Answer* grammar. A simple grammar should specify time, vehicle and departure stop. In addition, in should be able to represent line changes to the users. Since system responses are generated by linearizing a parse tree, the HTTP concrete does not need.

```
abstract Answer = DayTime, Stop ** {
flags
  startcat = Answer;
cat
  Answer;
  Vehicle;
  VhcTyp;
  Label;
  Tag;
fun
  Routing : Vehicle -> Stop -> Stop -> Time -> An-
swer;
  Change : Answer -> Answer -> Answer ;
  Vhc : VhcTyp -> Label -> Vehicle ;
  Lbl : Number -> Label ;
```

```
  Buss, Tram : VhcTyp ;
}
```

In this module vehicle labels are numbers. However, they can be specific names (e.g. Express, Red) that can be managed by grammar adaptation, which is described in chapter 5.

```
concrete AnswerEng of Answer = DayTimeEng, StopEng
** open (R=ResStop) in {
flags
  coding = utf8 ;
lincat
  Ans, Vehicle, VhcTyp, Label, Tag = { s: Str};
lin
  Routing = mkRoute;
  Change r1 r2 = { s = r1.s ++ "then" ++ r2.s} ;
  Vhc vhc lbl = { s = vhc.s ++ lbl.s} ;
  Lbl n = { s = "number" ++ n.s};
  Buss = { s = "bus"} ;
  Tram = { s = "tram"} ;

oper
  mkRoute : { s: Str} -> R.TStop -> R.TStop -> TTime
-> { s: Str} =
    \vhc, from, to, time ->
      {s = "Take" ++ vhc.s ++
           "from" ++ from.s  ++ from.t ++
           "to" ++ to.s ++ to.t ++
           "at" ++ time.s};
}
```

2.5.6 Travel Grammar

The travel module extends both *Query* and *Answer* modules. The main feature of this module is putting the *Answer* and *Query* categories in one category, which is the start category for parsing and dialogue generation. Furthermore, putting all grammars in one module allows the GF grammar to be adapted. As described in chapter 3, the extension grammar should extend one module that holds all grammar features.

```
abstract Travel = Query, Answer ** {
```

```
flags
  startcat = Stmt;
cat
  Stmt ;
fun
  Ask : Query -> Stmt;
  Reply : Answer -> Stmt;
}
```

The linearization rules of the Travel module are demonstrated below.

```
concrete TravelEng of Travel = QueryEng, AnswerEng,
**{
lincat
  Stmt = {s : Str} ;
lin
  Ask q = q;
  Reply p = p;
  Customize d = d;
}
```

2.6 Example Interaction

The system allows dialogues such as the example below. In the following dialogues, all phases of this dialogue are demonstrated in more details. This is a possible user query when English is selected as a language of conversation:

— U: I want to go from Chalmers to Valand today at 11:30

Given above query to the parser, this parse tree is produced:

(Ask

((((GoFromTo St_1597)

St_1667)

Today)

((HourMin ((Nums N1) (Num N1)))

((Nums N3) (Num N0)))))

From this, the linearizer generates the following HTTP GET request:

```
date= 8 &time= 1 1 : 3 0
  &originId= 9022014001960001
  &destId= 9022014007220001
```

As it is shown above date value is not in a standard format. Hence, some refinements are done on this translation to change the request to an acceptable format for the transport web service. After that, the base URL, service name and authentication key are attached to the translation.

```
http://api.vasttrafik.se/bin/rest.exe/v1/trip?
  authKey=734816b0- . . .-950b7a33337a
  &date=2012-5-19
  &time=11:30
  &originId=9022014001960001
  &destId=9022014007220001
```

Calling the web service by this GET request, An XML file is delivered. Then the dialogue manager extracts possible journeys and chooses the best journey to represent to the user. Subsequently, the corresponding parse tree is generated.

```
(Reply
      ((((Routing ((VhcLbl Tram) (Lbl (Num N7))))

                  St_1605)

                  St_1634)

                  ((HourMin ((Nums N1) (Num N1)))

                              ((Nums N3) (Num N1)))))
```

Finally, this parse tree is linearized to speech output:

— S: Take tram number 7 from Chalmers track A to Valand track A at 11:31

2.7 Component Overview

The dialogue system is integrated with a number of components which follow:

- GF software system. To compile multi-lingual grammars and generate PGF file.
- Java PGF interpreter. The JPGF library to work with PGF files and provide parser and linearizer.
- GF Writer. The application to generate or edit GF modules.
- Transport web service. The public transport web service to find real travel plan.
- Speech Synthesizer. The eSpeak speech synthesizer that converts text to speech.

Chapter 3

GF Grammar Adaptation

3.1 Introduction

GF can be used as a component in controlled natural language systems like dialogue systems, spoken translators and user interfaces. In most of these systems users require to adapt the system with their own needs. Consequently, GF grammars need to be updated when it is required. To reach this goal, GF grammars should be modified efficiently during execution of a system.

GF grammar adaptation allows users to adapt their system in different languages. Our solution is targeted toward adding new rules and also updating the definition of existing functions. In this chapter, the approach for adapting GF grammars is described.

3.2 Grammar Overview

As far as time is concerned, GF grammar adaptation can be costly while performing these two tasks: Modifying GF modules and reproducing the PGF file. Firstly, the GF system can only parse utterances that match grammar rules. Therefore adaptation needs changes in GF modules. Module modification needs a sequence of time consuming tasks, such as opening GF file, searching through rules and writing new rules. Moreover, user adaptations may cause changes not only in one module, but also in different modules. Accordingly, the modification process can be inefficient when changes need to be applied for several modules. Similarly, file size impacts on the speed of opening and updating. All in all, number and size of modules impacts on the cost of GF module modification.

Secondly, compiling GF grammars and producing new PGF file during execution takes some time and can be annoying for users. Since a PGF file is the

linkage of GF object files (.gfo files), more modified files causes more new GF object files and consequently creation of new PGF file would be more costly.

Regarding these problems, our approach is targeted toward applying changes to an extension grammar rather than changing the main grammar itself. The extension grammar extends all other modules and thus it contains all categories and functions of the main grammar. In addition to an abstract module, a concrete module is created for each language in extension grammar. Figure 8 describes the relation of the extension and main grammar. Using this approach, users can adapt grammars by both adding new grammar rules and updating existing rules.

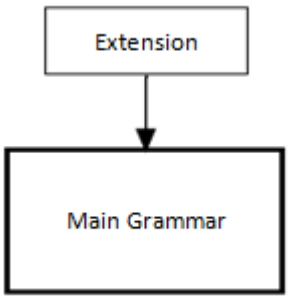

Fig. 8. GF grammar adaptation pattern design

Adapting GF grammars is possible by using GF Writer, which is described in chapter 1. Some usage and examples of GF grammar adaptation is shown in chapter 5, which are in a real dialogue system and with complicated grammar structure.

3.3 Adding New Functions

In order to add a new function definition or linearization to a predefined GF grammar, desired rules should be written in the extension abstract and concrete modules. Given a list of stop names, a new stop name will be added to the extension module like the example below. According to this example, `Stop_new` function is declared in the *Ext* abstract module and linearized in the *ExtEng* concrete module.

```
abstract Ext = Stop ** {
fun
```

```
  Stop_new: Stop;
}

concrete ExtEng of Ext = StopEng ** {
flags coding = utf8 ;
lin
  Stop_new = { s = "new stop" };
}
```

3.4 Updating Current Functions

A user can update an existing rule in GF grammar by either changing or adding alternatives to a function linearization. For both cases, exclusive inheritance must be used to avoid ambiguity. The following example illustrates how our *ExtEng* module has changed by adding an alternative to a predefined function.

```
concrete ExtEng of Ext = StopEng – [Chalmers]** {
flags coding = utf8 ;
lin
  Chalmers = StopEng.Chalmers | "university"
  Stop_new = "new stop";
}
```

As it is shown above, updating a function linearization causes changes just in the concrete module. Additionally, projection (`StopEng.Chalmers`) is used to keep the previous linearization of the function. A user should also notice that new terms (e.g. `university`) in the variant list must be of the same type.

Chapter 4

Example: An Adaptive Online Query System

4.1 Introduction

We have explained that a user in our multilingual dialogue system can find his travel plan by defining travel features, such as stop names, day and time. But most users want to have their own configuration and use shorter dialogues to demonstrate their means. For instance, a user may want to refer to a stop as “home” in his query. In order to address this issue, we propose a solution to support these user adaptations to the dialogue system explained in chapter 2. This solution is an example of GF grammar adaptation described in chapter 3.

Applying adaptation to the system allows users to add new definitions that can be used in future utterances. The functionality of this system is shown in the following example. First the user explicitly defines what he means by a special phrase and then he can act in his own interest by using customization in his queries.

— U: work means Chalmers on Monday at 7:30

— U: home means Valand

— U: I want to go from home to work

— S: Take tram number 10 from Valand track B to Chalmers track B at 07:33

Our approach for adapting the dialogue system provides users the ability to adapt their system in different languages. It is also targeted toward customizing each category and a number of categories in GF grammar. Furthermore, it provides users the ability to adapt their system for all supported languages while he configures the system for one language.

In addition to user adaptation, the dialogue manager sometimes needs to update GF grammar. One such situation is when a system requires new rules

to parse or linearize a phrase. For instance, in a travel planning system any changes may happen and the dialogue manager should be flexible enough to support all these changes. For instance, in our transport dialogue system new vehicle labels may be used in the system response. In this situation, the dialogue manger must support GF adaptation to add new definitions to the grammar.

In this chapter, we explain how an ordinary dialogue system can be changed to an adaptable one which supports both user and system adaptation. Here we use the transport dialogue system described in chapter 4.

4.2 Grammar Overview

In order to extend the transport grammar to an adaptive one, two modules are added. As we mentioned in chapter 2, a separate GF module is designed to hold all grammar adaptation. In addition, a separate module is designed that contains grammar rules for user definitions. In other words, to parse the user's definitions, corresponding grammar rules should be added to the system. These two modules are shown as *Ext* and *Def* in figure 9. The *Ext* module is initially empty and grammar rules will be added gradually for every new definition. The *Def* module is intended as a lexicon which offers new words that can be added as new concepts to the system.

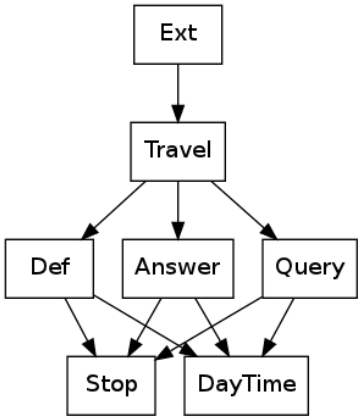

Fig. 9. Grammar design pattern for an adaptive transport dialogue system

According to figure 9, the *Ext* module inherits the content of all other modules. At first, abstract and concrete extension modules have no rules and new rules will be added gradually after user customizations.

```
abstract Ext = Travel ** {

}

concrete ExtEng of Ext = TravelEng ** {
flags coding = utf8 ;
}
```

In order to change our dialogue system to an adaptable one, GF grammars must be changed in some aspects. These changes are divided into two groups: dialogue manager and GF programmer changes. Dialogue manager changes means modifying the *Ext* abstract syntax and its concrete modules. These modifications are done dynamically in the dialogue manager and when a user defines new meanings. On the other hand, there are some changes that the GF programmer must consider in GF modules when writing the GF grammars. Adding type definitions, functions and operations are some of the examples. We explain these changes in the following sections in more details.

4.3 User Adaptation

In this section, it is explained how user adaptation is managed in the transport dialogue system. This user adaptation is based on the idea of voice programming where users can explicitly adapt some aspects of a dialogue system to their own needs [10]. Regarding voice programming, we describe user adaptation for four aspects of our transport system.

4.3.1 Definition Grammar

The *Def* grammar module specifies how a user can communicate to the dialogue system for the purpose of customization. We propose to adapt four aspects of the dialogue system. Accordingly, corresponding grammar rules are considered: `DefPlace`, `DefDay`, `DefPlaceDay`, `DefPlaceDayTime`. Moreover the user needs to define an alternative for a phrase definition. In the following abstract syntax, the type of these alternatives is similar, namely, `New`. For the purpose of brevity a simple syntax for this

grammar module is shown below. However, it should be large enough to support a wide variety of user utterances.

```
abstract Def = Stop, DayTime **{
flags startcat = Def ;
cat
  New;
  Def;
fun
  DefPlace : New -> Stop -> Def;
  DefDay : New -> Day -> Def;
  DefPlaceDay : New -> Stop -> Day -> Def;
  DefPlaceDayTime : New -> Stop -> Day -> Time ->
Def;

  Home, Work, Birthday, Weekend : New;
}
```

The English concrete syntax of the abstract syntax is demonstrated below. Since the user adaptation must be available for all supported languages, the Swedish grammar is also designed. It is not shown though.

```
concrete DefEng of Def = StopEng, DayTimeEng **{
flags
    coding = utf8;
lincat
  New, Def = {s : Str};
lin
  DefPlace new stop =
    {s = new.s ++ "means" ++ stop.s};
  DefDay new day =
     {s = new.s ++ "means" ++ day.s};
  DefPlaceDay new stop day =
     {s = new.s ++ "means" ++ stop.s ++ day.alt};
  DefPlaceDayTime new stop day time =
   {s = new.s ++ "means" ++ stop.s ++ day.alt ++
time.s};

  Home = {s = "home"};
  Work = {s = "work"};
  Birthday = {s = "birthday"};
  Weekend = {s = "weekend"};
```

```
}
```

As we mentioned before, the *Travel* grammar extends the *Query* and *Answer* grammars of this dialogue system. Similarly, it extends the *Def* module to cover all aspects of the dialogue system grammar. Therefore the following rules are added to the Travel grammar.

```
fun
  Customize : Def -> Stmt ;

lin
  Customize d = d;
```

4.3.2 Stop Customization

In this subsection we describe an example in our system for customizing the name of a stop. Firstly, the user defines a new command such as below:

— U: home means Valand

Then the parser produces corresponding parse tree.

(Customize

((DefPlace Home) St_1639))

Having this parse tree, the required information is extracted; that are *Home* and *St_1639*. After that, a new rule will be added to the concrete modules of *Ext*. Parallel to the English concrete syntax of *Ext*, the corresponding rule is added to the Swedish concrete syntax.

```
concrete ExtEng of Ext = TravelEng-[ St_1639 ] ** {
lin
  St_1639 = toStop TravelEng.Home
                   TravelEng.St_1639;
}
```

In order to apply the following adaptation to the Swedish language, all that needs to be done is to replace `Eng` term of above linearization rule to `Swe` in the *ExtSwe* module.

After adding a new linearization rule to the extension concrete module, the PGF file should be reproduced to support recent changes. Both module

modification and PGF file production are done using GF Writer methods. As it is shown below, the dialogue manager just needs two function names (`Home` and `ST_1639`) for `toStop` operation to generate a new linearization rule. Moreover, using `toStop` fanctor makes dialogue manager independet of GF grammar rules and type definitions.

```
//***** Modifying ExtEng module *****
String[] modules = { "TravelEng"};
String new = "Home";
String fun = "St_1639";
Element elem = new Element("St_1639",
  "toStop TravelEng." + new + " TravelEng." + fun);
Customizer.updateLin("./res/ExtEng.gf",
    modules, elem);

//***** Creating PGF file *****
List<String> travelBase = new LinkedList<String>();
travelBase.add("./res/ExtEng.gf");
travelBase.add("./res/ExtSwe.gf");
travelBase.add("./res/ExtHttp.gf");
Customizer.makePGF(travelBase, "./res/Ext.pgf");
```

After executing this java code, a new PGF file is produced. Since this PGF file holds new definitions, the user can ask queries containing these definitions in both English and Swedish:

— I want to go from Chalmers to home on Monday at 10:20

— Jag vill åka från Chalmers till hem på söndag kl 10:20

Travel Grammar Changes

There are some differences between the *Travel* grammar, illustrated in chapter 4, and the grammar we used for adaptation. These changes must be applied to the grammar by the GF programmer before releasing the dialogue system. Here we explain these changes in detail.

According to the ExtEng module shown above, we used restricted inheritance that excludes `St_1639` from *TravelEng*. This design causes non-ambiguity and simultaneously keeps the previous type definition of this function by using the `toStop` operation. The purpose of this operation is to change the type of user's alternative to the desired category, ex. String

(home) to Stop (Valand). The `toStop` operation is a part of the *TravelEng* module which is written before by the GF programmer.

```
concrete TravelEng of Travel = QueryEng, AnswerEng,
DefEng ** open (R=ResStop) in{
. . .

oper
toStop : {s : Str} -> Stop -> Stop =
    \new, stop -> {s = stop.s;
                   r = stop.r ;
                   t = stop.t;
                   alt = stop.alt | new.s};
. . .
}
```

As it is shown in the `toStop` operation, the type of `Stop` has changed in comparison to chapter 4, by adding `alt` object. This object contains a linearization of the stop name and its alternatives in the form of variants. The following lines of code show new type definition of Stop in our transport dialogue system. Notice the initial value of the `alt` object.

```
mkStop : Str -> Str -> Str -> TStop =
    \stop, region, track ->
      {s = stop; r = region; t = track;
       alt = stop ++ track};
```

Adding the `alt` object in the type definition allows GF programmers to use these alternatives whenever they need. For instance, we use the `alt` object for linearizing user queries but not the system response. The reason for this is that when the user listens to a travel plan, he prefers to know stop names rather than alternatives, such as home, work, etc.

```
GoFromTo from to day time =
  {s = "I want to go from" ++ from.alt ++
       "to" ++ to.alt ++
       day.alt ++
       "at" ++ time.s };

Routing vehicle from to time =
      {s = "Take" ++ vehicle.s ++
           "from" ++ from.s  ++ from.t ++
           "to" ++ to.s ++ to.t ++
```

```
            "at" ++ time.s};
```

Multiple Definitions for a Stop

The user may define several meanings for a special place. For instance, he defines Valand as both home and gym.

— U: gym means Valand

Since the ExtEng module has already a linearization for this stop, this alternative will be added as a variant to the linearization rule.

```
concrete ExtEng of Ext = TravelEng-[ St_1639 ] ** {
lin
  St_1639 = toStop TravelEng.Home
                   TravelEng.St_1639
          | toStop TravelEng.Gym
                   TravelEng.St_1639;
}
```

4.3.3 Day Customization

The approach for customizing day of a travel is similar to stop customization. Assuming this user command

— U: weekend means Sunday

The following parse tree is generated.

(Customize

((DefDay Weekend) Sunday))

Then, the dialogue manager executes the corresponding java codes to add this rule to the extension concrete modules.

```
concrete ExtEng of Ext = TravelEng-[ Sunday, . . .]
** {
lin
  Sunday = toWeekDay TravelEng.Weekend
                     TravelEng.Sunday;
. . .
```

```
}
```

The `toWeekDay` operation is an operation of the *TravelEng* module that adds an alternative to a given day.

```
toWeekDay : {s : Str} -> TDay -> TDay =
   \new, day ->
       {s = day.s;
        prep = day.prep;
        alt = day.alt | new.s };
```

Similar to the *Stop* category, a new object is considered in the *Day* type definition. Adding `alt` object to the type definition of an adaptable grammar is an initiative that separates original values from alternatives. The following shows the `alt` object of the *Day* category and its initial value.

```
oper
  TDay = {s : Str; prep : Str; alt : Str };

  mkDay : Str -> Str -> TDay =
       \d,p -> {s = d;
                prep = p;
                alt = p ++ d};
```

4.3.4 Day and Stop Customization

In the previous subsections we mentioned how an existing function in the GF grammar can be modified to hold user adaptations. In addition to this type of adaptation, the user may need to define an alternative for complicated phrases. In our transport dialogue system, this definition can be any combinations of stops, day, and time. To handle these definitions we need to introduce new types and consequently some operations. We describe our solution for this type of adaptation in the following example, where a user defines a word to mean a special day and place. A user utterance and its parse tree are like this:

— U: work means Chalmers on Monday

(Customize

(((DefPlaceDay Work) St_1597) Monday))

After extracting the required information from the parse tree, corresponding rules are added to the extension abstract and concrete modules. In other words, the dialogue manager adds a function definition to the *Ext* module and its linearization to the *ExtEng, ExtSwe and ExtHttp* modules.

```
abstract Ext = Travel ** {
fun
  WorkStopDay : StopDayTime;
  . . .
}

concrete ExtEng of Ext = TravelEng-[. . .] **{
lin
  WorkStopDay = toStopDay TravelEng.Work
                          TravelEng.St_1597
                          TravelEng.Monday;
. . .
}
```

The linearization rule for the Swedish syntax is similar to the English one, except for the *TravelEng* term that must be changed to *TravelSwe*. However, the linearization of WorkStopDay in ExtHttp module is different from ExtEng and that is due to omission of the `alt` abject from the `Stopday` type.

```
concrete ExtHttp of Ext = TravelHttp **{
lin
  WorkStopDay = toStopDay TravelHttp.St_1597
                          TravelHttp.Monday;
  . . .
}
```

In order to change the abstract and concrete modules, the dialogue manager needs to call two functions of GF Writer; that are `addFun` and `addlin`:

```
String new = "Work";
String stop = "St_1597";
String day = "Monday”;

//***** Modifying abstract module *****
Element elemAbs = new Element(new + "StopDay",
        "StopDay");
Customizer.addFun("./res/Ext.gf", elemAbs);
```

```
//***** Modifying English Concrete module *****
Element elemEng = new Element(new + "StopDay",
        "toStopDay TravelHttp." + stop +
        " TravelHttp." + day);
Customizer.addLin("./res/ExtEng.gf", elemEng);
```

After changing the GF modules, a new PGF file is created and the new customization is recorded in the dialogue system.

Travel Grammar Changes

In order to allow users to define a new meaning for a series of information that are not belonged to a category, a new category must be defined in the grammar. For instance, the `StopDay` category is declared in the *Travel* module to support user adaptation for this aspect of the dialogue system. Since `Stop` and `Day` are already declared in the grammar categories, the `StopDay` category is introduced to hold a stop and day together with a string as an alternative.

```
concrete TravelEng of Travel = QueryEng, AnswerEng,
DefEng ** open (R=ResStop) in{
lincat
  StopDay = {stop : R.TStop;
             day : TDay;
             alt : Str};
  . . .
}
```

Similar to the previous subsections, we need an operation to create a `StopDay` type from a given string, stop and day:

```
concrete TravelEng of Travel = QueryEng, AnswerEng,
DefEng ** open (R=ResStop) in{
oper
  toStopDay :
    {s : Str} -> R.TStop -> TDay ->
    {stop : R.TStop; day : TDay; alt:Str} =
      \ new, st, d ->
        {stop =st; day = d; alt = new.s};
  . . .
}
```

In addition to the English grammar, the HTTP grammar should also have the `toStopDay` operation and type definition for the `DayTime` category. Since user definitions are not used in HTTP queries, the `alt` abject is omitted.

```
concrete TravelHttp of Travel = QueryHttp, An-
swerHttp ** open (R=ResStopHttp) in {
lincat
  StopDay = { stop : R.TStop; day : TDay};
  . . .
oper
  toStopDay :
    R.TStop -> TDay ->
    {stop : R.TStop; day : TDay} =
      \st, d -> { stop = st; day = d};
  . . .
}
```

According to the *Ext* module, the `WorkStopDay` rule means having a certain stop name, day and phrase, so a new instance of *StopDay* type is produced. But it does not mean that the user can ask a query such as below.

— U: I want to go from Valand to work at 9:30

To address this issue, a new kind of `GoFromTo` function is introduced that accepts an instance of the `StopDay` category rather than separate `Stop` and `Day` instances.

```
fun
  GoFromToStopDay = mkQueryStopDay;

oper
  mkQueryStopDay :
    TStop ->
    {stop : TStop; day : TDay; alt : Str} ->
    TTime -> { s : Str} =
      \ from, stopDay, time ->
        {s = "I want to go from" ++ from.alt ++
             "to" ++ stopDay.alt ++
              time.s };
```

The corresponding function in TravelHttp module is shown below. Since HTTP requests need exact information for the users' query, all fields of `WorkStopDayTime` is used in linearization.

```
fun
  GoFromToStopDay from stopDay time =
    {s = "date=" ++ stopDay.day.s ++
         "&time=" ++ time.s ++
         "&originId=" ++ from.s ++
         "&destId=" ++ stopDay.stop.s};
```

Multiple Definition for Stop and Day

As it is shown in the *Ext* module, a new function will be declared for each customization of stop and day. Due to the GF grammar syntax, each function name must be unique in the abstract syntax. Accordingly, we formulate the function name generation by combining the alternative (e.g. Home) and *StopDay*. As a consequence, when a user defines a new definition for an existing function, the linearization will be changed to the new one. For instance, the `WorkStopDay` function will be the following when the user defines a new meaning for *work*.

— U: work means Chalmers Tvärgata on Monday

```
lin
  WorkStopDay = toStopDay TravelEng.Work
                          TravelEng.St_1590
                          TravelEng.Monday;
```

4.3.5 Time, day and Stop Customization

In this Section, we describe our approach for customizing a series of information containing literals. For example, time is a literal that is used in user queries in our dialogue system. When a user utterance is targeted toward defining an alternative for a specific stop, day and time, the related parse tree will be generated:

— U: work means Chalmers on Monday at 7:30

(Customize

((((DefPlaceDayTime Work)

St_1592)

Monday)

((HourMin (Num N7)) ((Nums N3) (Num N0)))))

Afterwards, the `WorkStopDayTime` function and its linearization are added to the *Ext* abstract and concrete modules.

```
fun
  WorkStopDayTime : StopDayTime;

lin
  WorkStopDayTime = toStopDayTime TravelEng.Work
                                  TravelEng.St_1592
                                  TravelEng.Monday
                                  "7 : 3 0";
```

The type definition of StopDayTime, holds stop, day, time and an alternative for combination of these elements. As it is shown below, type of `time` in the `StopDayTime` record is String and not `TTime`. This is due to the fact that time is similar in all languages and is not translated between languages. Moreover, the parse tree of a specified time is more than one function and consequently a complicated parse tree cannot be placed in the StopDayTime record.

```
StopDayTime = {stop : TStop;
               day : TDay;
               time : Str;
               alt : Str};
```

The `ToStopDayTime` operation and the GoFromToStopDayTime function are placed in the Travel concrete modules.

```
concrete TravelEng of Travel = QueryEng, AnswerEng,
DefEng ** open (R=ResStop) in{
lin
  GoFromToStopDayTime = mkQueryStopDayTime;

oper
  mkQueryStopDayTime :
    R.TStop ->
```

```
    { stop : R.TStop; day : TDay;
      time : Str; alt : Str} ->
    { s : Str} =
      \ from, stopDayTime ->
       {s = "I want to go from" ++ from.alt ++
            "to" ++ stopDayTime.alt };

toStopDayTime : {s : Str} ->
  R.TStop -> TDay -> Str ->
  {stop : R.TStop; day : TDay;
   time : Str; alt : Str} =
     \ new, st, d, t ->
       { stop = st; day = d; time = t; alt = new.s};

. . .
}
```

Similar to the previous section, the `alt` object is not needed for the `StopDayTime` category in the HTTP grammar. Accordingly, the `toStopDayTime` operation and the `GoFromToStopDayTime` function are without the `alt` object.

4.4 System Adaptation

In addition to user adaptation, the dialogue manager can update GF grammars. This allows the dialogue manager to adapt the system to new situations and keep the system always updated. We show the usage of system adaptation by an example in our transport dialogue system.

4.4.1 Vehicle Label Customization

Vehicle labels in our dialogue system are represented by the type of the vehicle (e.g. bus, tram) and a number, like bus number 10.

```
fun
  Vhc : VhcTyp -> Label -> Vehicle ;
  Lbl : Number -> Label ;
  Buss, Tram : VhcTyp ;
```

Having this grammar, we assume the following query:

— U: I want to go from Delsjömotet to Berzeliigatan today at 1 1 : 3 0

For this request, the web service offers Grön express bus. Since this type of vehicle label is not supported in the system grammar, the dialogue manager will fail to produce parse tree and the linearizer cannot generate system response.

To solve this problem, the grammar should cover all these exceptions for the vehicle label. Since there is no source to list these special vehicles, the system may encounter an exception anytime. The only feasible solution for this problem is to use GF grammar adaptation and add the desired vehicle label to the extension grammar.

When travel planner offers a vehicle with specific name, the dialogue manger checks whether this label is already added to the grammar or not. This is done by parsing the vehicle label. If the parser succeeds to parse the vehicle's name, the function name will be used in the answer parse tree. Otherwise, a new function will be added to the extension abstract and concrete modules.

```
abstract Ext = Travel ** {
fun
  Lbl_20120504_0047 : Label;
  . . .
}

concrete ExtEng of Ext = TravelEng-[ . . . ]** {
flags coding = utf8 ;
lin
  Lbl_20120504_0047 = toLabel "GRÖN EXPRESS";
 . . .
}
```

According to above grammar, a combination of system day and time is used for generating unique function names.

After adding new labels to the grammar, the PGF file is updated. Afterwards, the parse tree and its linearization will be generated:

(Reply
(((((Routing ((Vhc Buss) **Lbl_20120504_0047**))
St_2266)
St_1734)
((HourMin ((Nums N1) (Num N1)))

((Nums N4) (Num N1)))))

— S: Take bus Grön express from Delsjömotet to Berzeliigatan at 1 1 : 4 1

Chapter 5

Evaluation

Since user adaptation results more accurate and shorter input queries, speech recognition is more robust and with fewer errors. In order to evaluate this assessment, we tested 120 random generated input queries of our transport query system. These utterances were equally divided into two groups of adapted and non-adapted queries and were fed to the speech recognizer, which was Google speech recognizer[3]. After collecting the outputs of the Google speech recognizer we noticed that all of the non-adapted queries failed, whereas most of the adapted queries were passed.

The behavior of the speech recognizer while encountering foreign words is shown in this typical example:

— Input: I want to go from **Åketorpsgatan to Billdal** on Monday at 11:31
— Output: I want to go from **pocket doors car tom to build on that** on Monday at 11:31

According to this example, the speech recognizer's trend is to find known words instead of analyzing foreign words. In other words, it extracts a sequence of common words rather than guessing the given place name; so it cannot translate even a plain name such as Billdal. However, having look to the failed adapted queries demonstrates that the recognized text is very similar to the speech and the error rates are low. For instance, in the following query the word "pub" is translated to "park", which is due to the difficulty of discriminating between special alphabets.

— Input: I want to go from the **pub** to park on Friday at 15:35
— Output: I want to go from the **park** to park on Friday at 15:35

In contrary, the following examples show some adapted passed queries:

[3] http://www.google.com/insidesearch/features/voicesearch/index-chrome.html

— I want to go from hospital to restaurant
— I want to go from bank to cinema at 6:17
— I want to go from university to university on Monday at 4:20

In order to evaluate and compare the word error rate of speech recognizer for both adapted and non-adapted queries, the similarity of each query to the recognized one was numerated word by word and in a sequential order. We chose this type of similarity according to the GF parser, which parses a given sentence token by token and using a top-down algorithm [11]. The following table shows the rates of sentence error and word error for both adapted and non-adapted queries:

	Word error rate	Sentence error rate
Non-adapted queries	58	100
Adapted queries	26	53

Table 1. Error rate of speech recognizer for adapted and non-adapted queries

To sum up, user adaptation affects the speech recognition process and results to more reliable system. This is due to the elimination of foreign words and using shorter dialogues. By user adaptation, stop names are replaced with user defined synonyms and consequently they can be recognized with higher probability.

Chapter 6

Conclusion

6.1 Contributions

In this work, we introduced an adaptation technique for dialogue systems that are based on Grammatical Framework (GF). The baseline system for demonstrating the adaptation technique is a GF-base query system for planning journeys. This multi-lingual travel planning system can present up-to-date travel plans by communication with a transport web service. In order to use this system for a new transport network, the GF modules that holds stop names, are generated automatically by the *GF Writer* application.

The embeddable GF writer application can produce or update GF grammars during the execution of a program. This application can also compile the modified GF modules and generate a new PGF files by running GF software commands. Moreover, it makes a major contribution in the implementation of the adaptation technique.

The adaptation technique is based on placing grammar changes in an extension grammar rather than changing the main grammar itself. This extension grammar extends the main grammar and thus it contains all grammar rules of the main grammar. Since only the extension grammars needs to compile after each adaptation, the execution time for generating a new PGF file is short.

We applied this adaptation technique to our base line system to have an adaptable transport query system. This adaptable system shows some examples of this technique for both user adaption and system adaptation. By user adaptation, the users can adapt some aspects of the transport query system to their own needs, such as defining alternatives for stop names, day and time of a travel. By system adaptation, the system can automatically add the name of a newly added bus line to the system grammar.

This adaptable transport system shows how users can customize the system for their own purposes. Since user adaptation results more accurate and shorter input queries, speech recognition will be more robust and with fewer errors. This assessment was evaluated by 120 random queries and it turns out that speech recognition is more reliable in adaptable systems due to elimination of foreign words and using shorter queries.

6.2 Application Source Code

The source code of the application which is written as a part of this work is available from `https://github.com/hasibi/DynamicTravel`. The code consists of the embedded GF writer application and the travel planning query system.

Chapter 7

Future Work

7.1 Adaptation Technique

This adaptation technique can be applied to various domains of information-seeking dialogue systems in which the dialogue system integrates a huge database, a lexicon and a set of dialogue plans.

For instance, the SAMMIE [17] is an in-car dialogue system for a music player application that allows users to control the currently playing song, construct an edit playlists [17] [18]. In this system, the users interact with the system using lots of foreign words to look for songs, artists, albums and etc. Applying user adaptation to this dialogue system, allows the users to define alternatives for a set of foreign words or to shorten the length of utterances. Consequently, these laconic conversions will increase the driver's attention to the primary driving task.

Another example is price comparison services such as PriceRunner[4], where users define features of a specific product to compare between different retailers. A dialogue system in this domain needs to record the features of a product as a special name which can be used in later utterances. For instance, when the user wants to check the price of a particular camera regularly, he can define a synonym (e.g. my camera) for a special camera rather than repeating the camera model every time.

7.2 Transport Query System

This text-based query system can be extended to a multimodal dialogue system [8] to support speech, text and map clicks. In such system, the users can use text or map clicks to define alternatives for stop names which cannot be

[4] http://www.pricerunner.co.uk/

recognized by speech recognizer. After that, the user can refer to that place by using that alternative in later dialogues. Moreover, the user adaptations can be saved in a log file, which will be retrieved when the stop grammar changes.

Since all stop names are saved as a part of grammar, the system can support predictive parsing [19] to offer stop names to the users. Moreover, alternatives that are used in user adaptation are preferred to come from a bigger a lexicon. Resource Grammar Library (RGL) [15] can be used for this purpose to scale up user's dialogues. Additionally, the system grammar can be extended to support more dialogues and generate complex answers.

Due to the large amount of information that is presented to users in this travel planning system, the user may lose some information of the system's response. In order to improve the efficiency of information presentation phase, some approaches like UMSR (User Model Based Summarize and Refine) can be applied to this system [20]. In addition the user's customization can be used in the system response to make the dialogue shorter. For instance, instead of a special stop name, the system can use the user's definition for that stop.

7.3 GF Writer Application

Presently, the GF writer application can generate and modify most parts of an abstract or concrete module, like opening a resource, module extension and etc. The following is some extensions that make the GF Writer more general and applicable.

- Supporting operation definition in a concrete module.
- Generation and modification of resource module type.
- Adding more functionality for grammar modification, e.g. editing GF categories, deleting a specified rule and so forth.
- Using more efficient algorithm for both searching a function's name and applying appropriate action.

References

Angelov, K.., Bringert, B., and Ranta, A.:. Pgf: A portable runtime format for type-theoretical grammars. Journal of Logic, Language and Information, 19:201- 228, 2010. (n.d.).

Angelov, K.: Incremental Parsing with Parallel Multiple Context-Free Grammars. In: European Chapter of the Association for Computational Linguistics (2009). (n.d.).

Angelov, K.: The Mechanics of the Grammatical Framework. PhD thesis, Chalmers University of Thechnology and University of Gothenburg, 2011. (n.d.).

Becker, T., Poller, P., Schehl, J., Blaylock, N.,Gerstenberger, C., Kruijff-Korbayova, I.: "The SAMMIE system: Multimodal in-car dialogue". In Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions, pages 57–60, Sydney, Australia.

Bringert, B., Cooper, R., Ljunglöf, P., Ranta, A.,: Multimodal Dialouge System Grammars. In: Proceedings of DIALOR'05, Ninth workshop on the semantics and pragmatics of dialo-gue, Nancy, France, pages 53-60, June 2005. (n.d.).

Bringert, B.: Embedded grammars. MSc Thesis, Department of Computing Science, Chalmers University of Technology, 2004. (n.d.).

Bringert, B.: Speech recognition Grammar Compilation in Grammatical Framework. (n.d.). *In: Proceedings of the ACL 2007 workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic, June 29, 2007, pages 1-8, by the Association for Computational Linguistics.*

Demberg, V., Winterboer, A., Moore, J. D.: A Strategy for Information Presentation in Spoken Dialog Systems, Computational Linguistics 37(3): 455-488, 2011.

Georgila, K., Lemon, O.: Programming by Voice: enhancing adaptivity and robustness of spoken dialogue systems, In BRANDIAL'06, Proceedings of the 10th Workshop on the Se-mantics and Pragmatics of Dialogue, pages 199–200. (n.d.).

Hart, G., Johnson, M., Dolbear, C.: Rabbit: Developing a control natural language for au-thoring ontologies. In: ESWC. (2008) 348–360. (n.d.).

Martin-L□öf, P.: Intuitionistic type theory. Naples:. Bibliopolis, 1984. (n.d.).

Milner, R., Tofte, M., and Macqueen, D. The Definition of Standard ML. MIT Press, Cambridge, MA, USA,(1997). (n.d.).

Perera, N.,Ranta, A.: "Dialogue System Localization with the GF Resource Grammar Library". In SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague, 2007.

Ranta, A., Enache, R., Détrez, G.: Controlled Language for Everyday Use: the MOLTO Phrasebook, Controlled Natural Languages Workshop (CNL 2010). (n.d.).

Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011) ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). (n.d.).

Ranta,A.: The GF Resource Grammar Library.In: Linguistic Issues in Language Technology, LiLT 2:2, December 2009. (n.d.).

Schwitter,R.: Controlled Natural Languages for Knowledge Representation. In: Proceedings of 23rd International Conference on Computational Linguistics, Beijing, China, pp. 1113–1121 (2010). (n.d.).

Shiffman, R.N., Michel, G., Krauthammer, M., Fuchs, N.E., Kaljurand, K., Kuhn, T.: Writing clinical practice guidelines in controlled natural language. In: Proceedings of the 2009 conference on Controlled natural language. (n.d.). *CNL'09, Berlin, Heidelberg, Springer-Verlag (2010) 265–280.*

Thompson, S.: Haskell: The Craft of Functional Programming. Addison-Wesley, 2nd edition, (1999). (n.d.).

Wyner, A., Angelov, K., Barzdins, G., Damljanovic, D., Davis, B., Fuchs, N., Hoefler, S., Jones, K., Kaljurand, k., Kuhn, T., Luts, M., Pool, J., Rosner, M., Schwitter, R., Sowa, J.: On controlled natural languages: Properties and prospects. (n.d.). *In Norbert E. Fuchs, editor, Proceedings of the Workshop on Controlled Natural Language (CNL 2009), volume 5972 of Lecture Notes in Computer Science, pages 281–289. Springer, Berlin / Heidelberg, 2010.*

Appendix A

Methods of GF Writer

A.1 Introduction

The GF Writer is an embedded Java API that writes or updates GF grammars in a defined module.

This document represents the functionality of this document in detail.

A.2 Abstract Class

To generate a GF abstract module, required information should be passed to two functions: the constructor and the write method. The descriptions of class constructors are in accordance with the signatures which follow.

A.2.1 Constructors

```
Abstract (GFFilePath, Extend, Open, Flags, Cat, Fun)
Abstract (GFFilePath, Cat, Fun)
```

Parameters

GFFilePath. The module address. The address should contain the module name with *.gf* extension.

Extend. A list of extension modules. Each module of this list should be an instance of *ResInherit* class.

Open. A list of resource module names. These modules will be opened by created module.

Flags. A list of flag names and their values.

Cat. A list of categories in abstract syntax.

Fun. A list of function declarations.

Example Abstract Syntax

The example below shows a simple abstract syntax and related Java code of the GF Writer.

```
abstract Stop = {
cat
  Stop;
fun
  St_0 : Stop;
}
```

Java Code

```
List<String> cat = new LinkedList<String>();
cat.add(Stop);
List<Element> fun = new LinkedList<Element>();
fun.add(new Element("St_0", "Stop"));
Abstract abs = new Abstract("./Stop.gf", cat, fun);
abs.write();
```

A.3 Concrete Class

A.3.1 Constructors

```
Concrete(GFFilePath, Abs, Extend, Open, Flags, Lin-
Cat, Lin)
Concrete(GFFilePath, Abs, LinCat, Lin)
```

Parameters

GFFilePath. The module address. The address should contain the module name with .gf extension.

Abs. The name of abstract syntax that is used for this concrete syntax.

Extend. A list of extension module. Each module of this list should be an instance of *ResInherit* class.

Open. A list of resource module names. These modules will be opened by created module.

Flags. A list of flag names and their values.

LinCat. A list of type definitions.

Lin. A list of function linearizations.

Example Concrete Syntax

```
concrete StopEng of Stop = open ResStop in {
flags
  coding = utf8;
lincat
  Stop = {s : Str; r : Str};
lin
  St_0 = mkStop "Chalmers" "Göteborg" ;
}
```

Java Code

```
String rhs = "mkStop \"Chalmers\" \"Göteborg\"";
List<Element> lin = new LinkedList<Element>();
lin.add(new Element("St_0", rhs));

List<Element> lincat = new LinkedList<Element>();
lincat.add(new Element(Stop, "{s : Str; r : Str}"));

List<Element> open = new LinkedList<String>();
open.add("ResStop");

List<Element> flags = new LinkedList<Element>();
flags.add(new Element("coding", "utf8" ));

Concrete con = new Concrete("./StopEng.gf", "Stop",
null, open, flags, lincat, lin);
con.write();
```

A.4 Customizer class

A.4.1 Compiling GF grammars

The following function produces a PGF file for a given concrete modules with the same abstract syntax. It uses GF as a batch compiler and runs the *"make"* command: `gf –make SOURCE.gf`

```
static void makePGF(BaseGrammars, PGFFilePath)
```

Parameters

BaseGrammar. A list of GF concrete modules (with path) for compiling.

PGFFilePath. A path for generated PGF file.

Example

```
List<String> stops = new LinkedList<String>();
travelBase.add("./res/StopEng.gf");
travelBase.add("./res/StopSwe.gf");
Customizer.makePGF(stops, "./res/Stop.pgf");
```

A.4.2 Adding grammar rules

```
static void addLin(GFFilePath, Elem)
static void addFun(GFFilePath, Elem)
```

Parameters

GFFilePath. The path of a GF module.

Elem. A rule of GF grammar, either function declaration or linearization; that is an instance of the *Element* class.

Example

```
concrete StopEng of Stop = open ResStop in {
flags
  coding = utf8;
lincat
  Stop = {s : Str; r : Str};
Lin
  St_new = mkStop "newStop" "Göteborg" ;
  St_0 = mkStop "Chalmers" "Göteborg" ;
}
```

In order to add `st_new` rule to the existing *StopnEng* module, the following java code is needed.

Java Code

```
String rhs = "mkStop \"newStop\" \"Göteborg\"";
Element elemAbs = new Element("St_new", rhs);
Customizer.addFun("./res/StopEng.gf", elemAbs);
```

A.4.3 Updating a GF Rule by Inheritance

The following method changes the linearization of a function from another module. This module must use exclude inheritance and define a new linearization for the desired function. The main usage of this function is in GF grammar adaptation.

```
static void updateLin(GFFilePath, Module, Elem)
```

Parameters

GFFilePath. The path of a GF module that holds changes.

Module. The name of the GF module that contains desired function

Elem. A function linearization; that is an instance of *Element* class.

Example

```
concrete ExtEng of Ext = StopEngEng-[St_0]** {
flags coding = utf8 ;
lin
  St_0 = mkStop "NewChalmers" "Göteborg" ;
}
```

In order to update the linearization of `St_0` from *ExtEng* module, the following java code is needed. The `updateLin` method, will add exclusive inheritance header to the *ExtEng* module, if it is not already present. Moreover, if the `St_0` is already exists, it will replace the linearization with the new one.

Java Code

```
String rhs = "mkStop \"NewChalmers\" \"Göteborg\"";
Element elem = new Element("St_0", rhs);
```

```
Customizer.updateLin("./res/ExtEng.gf", "StopEng",
elem);
```