An Adaptation Technique for GF-Based Dialogue Systems

Faegheh Hasibi

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
faegheh@student.chalmers.se

Abstract. This paper introduces a technique for adapting GF (Grammatical Framework)-based dialogue systems. This technique can be used to adapt dialogue systems in two aspects: user adaptation and self-adaptation. By user adaptation, users can customize the system to their own needs and define alternatives for a series of information to use in later utterances. By self-adaptation, the system can update GF grammar to keep the system adequate when new situations occur. This technique is demonstrated by a multi-lingual transport query system, which allows users to find up-to-date travel plans. Adapting GF-based dialogue systems improves the functionality of speech recognizers by defining alternatives for specific phrases and also keeps the dialogue system always updated.

Keywords: Adaptation, user adaptive, self-adaptive, transport dialogue system, Grammatical Framework, Controlled Natural Language, travel planning, dialogue system, speech recognition.

1 Introduction

The notation of adaptivity is an important area in the spoken dialogue systems and concerns the natural manner of humans while interacting with a dialogue system [1]. These manners consist of different interaction styles, behavior, vocabulary and preferences [1]. For instance users may need to adapt the dialogue system to their own needs and communicate with the system by specific utterances. This paper presents a novel technique for adapting information-seeking dialogue systems in which the dialogue system integrates a huge database, a lexicon and a set of dialogue plans. This technique can extend an information-seeking dialogue system to either a self-adaptive or user-adaptive system and is applicable to various domains of controlled natural languages (CNLs).

For instance, the SAMMIE [2] is an in-car dialogue system for a music player application that allows users to control the currently playing song, construct an edit playlists [2, 3]. 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.

T. Kuhn and N.E. Fuchs (Eds.): CNL 2012, LNCS 7427, pp. 95–109, 2012.

Another example is price comparison services, such as PriceRunner[1], 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.

Similarly, the user adaptation can be implemented in transport dialogue systems, where the users ask for a travel plan. For instance, the Gothenburg Tram Information System (GOTTIS) [4] is a multilingual and multimodal transport dialogue system, which uses a weighted directed graph for finding the shortest path through a subset of the Gothenburg public transportation network [5, 6]. In such systems, the user can ask a travel plan by using an alternative instead of a specific bus/tram stop, date and time to have more concise dialogues.

The idea of extending dialogue systems by allowing users to reconfigure the system to their interest is represented by voice programming [7]. This kind of adaptation is simply done when users define their own commands in speech dialogues [7]. The demonstration technique allows user to adapt a dialogue system by using the idea of voice programming. In other words, the users can reconfigure the system by defining synonyms for expressions that will be used frequently in later conversations. This kind of adaptation will improve the speech recognition of dialogue system due to the elimination of foreign words and use of shorter dialogues. Moreover, it will result in concise and brief dialogues that will increase the satisfaction of users.

In addition to user adaptation, our technique can be used for self-adaptation, where the system can modify the lexicon of a dialogue system in response to changes in the system environment. Hence, self-adaptation is a solution to the problem of context change in online dialogue systems as it keeps the lexicon always updated. For instance, when new bus/tram lines are added to a transport network, the transport dialogue system should adapt itself by adding the new bus/tram lines to the lexicon.

In this paper we explain an adaptation technique for dialogue systems that are based on Grammatical Framework (GF) [8]. GF is a grammar formalism designed for supporting grammars of multi-lingual controlled languages. The key feature of GF is the distinction between two components of grammars: abstract syntax and concrete syntax. Every GF grammar has one abstract syntax and one or more concrete syntaxes. The abstract syntax defines which expressions can be built by the grammar and the concrete syntax describes how these expressions are linearized in a particular language.

The baseline system for demonstrating the adaptation technique is a GF-based query system for planning journeys, which is described in section 2. The method of dialogue system adaptation is described in section 3, and the usages of this technique are illustrated by some examples in section 4. These examples show how our technique can be used in dialogue systems to provide both user adaptation and self adaptation. Finally we evaluate our results and summarize our experiments in section 5 and 6, respectively.

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

2 The Baseline System

The baseline system is a multi-lingual GF-based query system which presents up-to-date travel plans. This system uses PGF [9] (Portable Grammar Format) grammars which are used as a target of compiling grammars written in GF. In order to work with PGF, a PGF interpreter is needed, which can perform a subset of GF system functionality, such as parsing and linearization. As 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. After that, travel information is retrieved from the web service response and the corresponding abstract syntax tree is generated and linearized to a natural language answer.

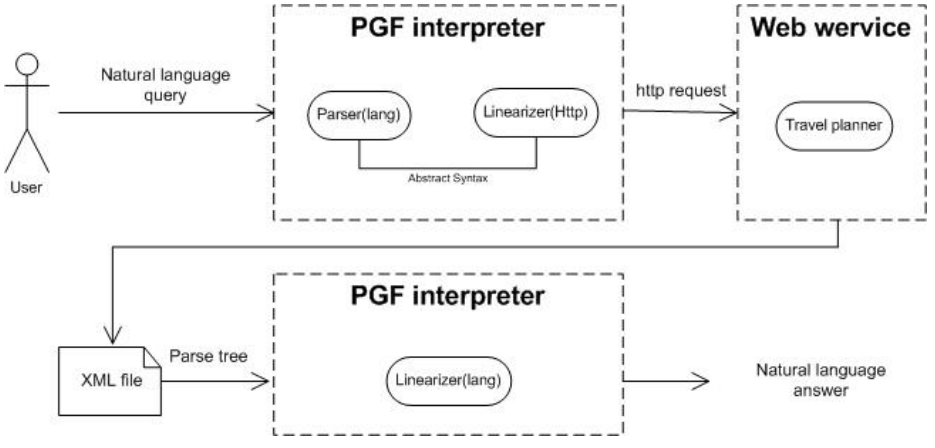

Fig. 1. Architecture overview of multilingual travel planning query system

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 and since the HTTP language and natural languages have the same abstract syntax, both parsing and linearization can be done for phrases in a new language.

The described system uses the Gothenburg transport web service[2] and supports queries in both English and Swedish. The following is an ordinary interaction between user and system:

- 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

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 hold stops, are generated automatically by the GF writer application, which will be introduced in the next subsection.

[2] `http://www.vasttrafik.se/`

2.1 The GF Writer Application

The embeddable GF Writer[3] 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 part of the grammar, such as module header and body, flags, categories and function declarations must be defined. Similarly, for updating a GF grammar, a module name and a new function definition are needed. After generating or modifying a GF grammar, a new PGF file will be generated and replaced with the previous one.

The GF Writer offers three main classes for creation and modification of the GF modules: *Abstract*, *Concrete* and *Customizer* class. The *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 *Customizer* class, grammars will be updated according to user's requests.

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 translator, dialogue systems and localization purpose applications. Moreover, this application makes a major contribution in the implementation of the adaptation technique.

2.2 System Grammar

The transport system contains a set of modules for query and answer utterances. Figure 2 depicts an overview of grammar module, produced with the module dependency visualization feature in GF. According to this figure, both *Query* and *Answer* grammars use the same modules for travel time and stop names and on top of these grammars, the*Travel* grammar extends both *Query* and *Answer* grammars to put all grammar rules together.

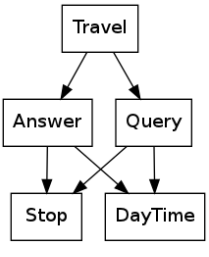

Fig. 2. Grammar design pattern for a transport query system

Stop Grammar. This grammar represents the bus/tram stops of the transport network. The straightforward approach for supporting stop names in the system grammar is to use string literals, but this *Stop* grammar is useful to suggest the stop names to the user by using the word prediction feature of the PGF parser, while parsing the user's queries.

[3] `https://github.com/hasibi/DynamicTravel`

The *Stop* modules are automatically generated by the GF writer application to make the system portable to other transport systems, which have the same format in HTTP requests and XML file. In order to generate the *Stop* grammar, the following tasks are performed in the transport query system:

1. Sending a query to transport web service to get a list of all bus/tram stops
2. Parsing XML file and retrieving the information of stops
3. Generating transport network modules for both natural languages and HTTP request

The generated abstract syntax offers a unique numerical function name for each stop. These numerical function names prevent the ambiguity of stops with the same names, which are different in other details (e.g. the same street in two different regions).

```
fun
  St_0 : Stop;
```

The English concrete module linearizes *Stop* terms in records with the name, region and track of each stop. Using this structure, the query functions are free to use the stop details (e.g. region, track number) or not.

```
concrete StopEng of Stop = {
lincat
  Stop = {s: Str; r: Str; t: Str};
lin
  St_0 = {s = "Valand"; r = "Göteborg"; t = "track A"};
  St_1 = {s = "Abacken"; r = "Skövde"; t = "track B"};
  . . .
}
```

The *StopHttp* concrete syntax contains the stops' 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 = {
lincat
  Stop = {s : Str};
lin
  St_0 = {s = "9022014004420003"};
  . . .
}
```

DateTime 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 or tomorrow, to refer to the date of the travel in a dialogue system. Regarding this fact, we encode each day to a number in *DayTimeHttp* module and replace it in HTTP request after calculating the corresponding date in our Java program. The following code is a part of the *DayTime* module related to the day of travel.

```
fun
  Today, Tomorrow, Monday, ... : Day;
```

In the English concrete syntax of the *DayTime* module, the indexical expressions are defined in the `Today` function.

```
lin
  Today = {s = "today"} | {s = ""};
```

This function means that the queries with no reference to a day are interpreted as current day. Accordingly, this feature adds both indexicality and context-awareness to the query system.

Query Grammar. The HTTP concrete module of the *Query* abstract module generates HTTP requests in collaboration with other concrete modules. The developed dialogue system uses Göteborg transport service as a travel finder. Since this web service supports the HTTP GET method, the `GoFromTo` function produces an HTTP request with respect to the user's query.

```
fun
  GoFromTo : Stop -> Stop -> Day -> Time -> Query ;

lin
  GoFromTo from to day time =
    {s = "date=" ++ day.s ++ "&time=" ++ time.s ++
         "&originId=" ++ from.s ++ "&destId=" ++ to.s};
```

Answer Grammar. The system response grammar is described in the *Answer* grammar, which consists of vehicle, departure stop and time of travel. Since the system responses are generated by linearizing a parse tree, the HTTP concrete can have no rules. In this system we used a simple grammar for natural language utterances; however for the mature system the Resource Grammar Libraries (RGLs) [10] can be used.

```
fun
  Routing : Vehicle -> Stop -> Stop -> Time -> Answer;
  Vhc : VehicleType -> Label -> Vehicle ;
  Lbl : Number -> Label ;
  Buss, Tram : VehicleType;

lin
  Routing vehicle from to time =
    {s = "Take" ++ vehicle.s ++
         "from" ++ from.s  ++ from.t ++
         "to" ++ to.s ++ to.t ++
         "at" ++ time.s};
```

Travel Grammar. The travel module extends both *Query* and *Answer* modules. The main feature of this module is putting *Answer* and *Query* category 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, which will be described in the next section.

```
abstract Travel = Query, Answer ** {
flags
  startcat = Stmt;
cat
  Stmt ;
fun
  Ask : Query -> Stmt;
  Reply : Answer -> Stmt;
}
```

3 The Adaptation Technique

GF can be used as a component in controlled natural language systems, such as dialogue systems. To make these systems adaptive, the GF grammars need to be updated when it is required and to reach this goal, GF grammars should be modified efficiently during execution of the system.

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 modules must be modified since the GF system can only parse utterances that match grammar rules. Module modification needs a sequence of time consuming tasks, such as opening a 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 and the situation can be even worse when the GF module is huge.

Secondly, compiling GF grammars and producing new PGF files during execution of a program 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 the creation of a new PGF file would be more costly. All in all, adapting GF grammar by changing the grammar modules is time consuming and will be done in an unacceptable time for users.

Regarding these problems, our approach is targeted toward applying changes to an extension grammar rather than changing the main grammar itself. The extension grammar is a GF module that extends all other grammars and thus it contains all categories and functions of the main grammar. According to the separate compilation feature of the GF software, the only grammar module that is needed to compile after every adaptation will be the extension module. So the execution time of the GF grammar adaptation becomes acceptable for the users. Figure 3 describes the relation of the extension and main grammar.

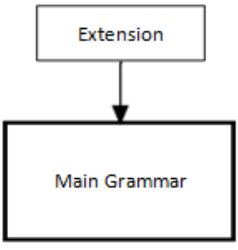

Fig. 3. GF grammar adaptation pattern design

For instance, to extend a GF grammar to an adaptive one, two grammars are needed: *Ext* and *Def*. The *Ext* module is initially empty and grammar rules will be added gradually for every new definition. The *Def* module contains grammar rules for user definitions. These two modules are shown in figure 4.

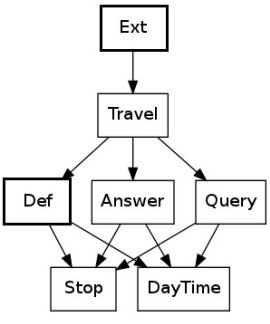

Fig. 4. Grammar design pattern for an adaptive transport dialogue system

In order to change our dialogue system to an adaptable one, the 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 *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 while writing the GF grammars. Adding type definitions, functions and operations are some of the examples. We explain these changes in the following subsections in more details.

This adaptation technique can be used for both self adaptation and user adaptation. These two types of adaptation are explained by some examples in the following subsections.

3.1 User Adaptation

The 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 [7]. Regarding voice programming, the user adaptation is described for two aspects of the transport system.

Definition Grammar. The *Def* grammar specifies how a user can communicate to the dialogue system for the purpose of customization. A simple syntax of the *Def* grammar is shown below, which offers some synonyms to the user for defining a new meaning for a series of information.

```
fun
  DefPlace : New -> Stop -> Def;
  DefPlaceDayTime : New -> Stop -> Day -> Time -> Def;
  Home, Work, Gym : New;

lin
  DefPlace new stop = {s = new.s ++ "means" ++ stop.s};
  DefPlaceDayTime new stop day time =
   {s = new.s ++ "means" ++ stop.s ++ day.s ++ time.s};
```

Stop Customization. In this subsection, an example of customizing the name of a stop is described. Firstly, the user defines a new command such as below:

— U: home means Valand

Then the parser produces corresponding abstract syntax tree.

(Customize
((DefPlace Home) St_1639))

Having this parse tree, the required information is extracted; *Home* and *St_1639*. In the next step, a new rule is added to the concrete modules of *Ext*. Parallel to the English concrete syntax of *Ext*, a corresponding rule is added to the concrete syntax of other languages.

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

After adding the new linearization rule to the extension concrete module, the PGF file is reproduced to support recent changes. Both the module modification and the PGF file production are done using GF Writer methods.

Travel grammar changes. There are some changes that must be applied to the grammar by GF programmer before releasing the dialogue system. In 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 `toStop` operation. The purpose of this operation is to change the type of user's alternative to the desired category, e.g. String (home) to Stop (Valand). The `toStop` operation is a part of *TravelEng* module which is written by the GF programmer.

```
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 with section 2, by adding the `alt` object. This object contains a linearization of the stop name and its alternatives in the form of variants.

```
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.

```
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 names for a special place. For instance, he defines Valand as gym in addition to home.

— 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.

```
St_1639 = toStop TravelEng.Gym (TravelEng.Home
                                TravelEng.St_1639);
```

Stop, Day and Time Customization. In the previous subsection 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 place, day and time. A user utterance and its parse tree are like this:

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

(Customize
((((DefPlaceDayTime Work) St_1592) Monday)
((HourMin (Num N7)) ((Nums N3) (Num N0)))))

As the next step, 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
                                     "7:30";
```

Travel grammar changes. Since `Stop`, `Day` and `Time` are already declared in the grammar categories, the `StopDayTime` category is introduced to hold a stop, day and time together with a string as an alternative. As it is shown below, type of `time` in the `StopDayTime` record is String and not `Time`. This is due to the fact that time is similar in all languages and is not translated between languages.

```
StopDay = {stop: Stop; day: Day; Time: Str; alt: Str};
```

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

```
toStopDayTime: {s:Str} -> Stop -> Day -> Str -> StopDay =
  \new, st, d, t ->
  {stop = st; day = d; time = t; alt = new.s};
```

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

```
Lincat  StopDayTime = {stop : R.TStop; day : TDay};

oper toStopDayTime : Stop -> Day -> Str -> StopDaTime =
     \st, d, t -> { stop = st; day = d; time = t };
```

According to *Ext* module, the `WorkStopDayTime` rule means having a certain stop name, day and phrase, a new instance of *StopDayTime* 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 `StopDayTime` category rather than separate `Stop, Day` and `Time` instances.

```
GoFromToStopDayTime from stopDayTime =
  {s = "I want to go from" ++ from.alt ++
       "to" ++ stopDayTime.alt };
```

The corresponding function in TravelHttp module is shown below. Since the HTTP requests need exact information for the user's query, all fields of `WorkStopDay-Time` are used in linearization.

```
GoFromToStopDayTime from stopDayTime =
  {s = "date=" ++ stopDayTime.day.s ++
       "&time=" ++ stopDayTime.time.s ++ "&originId=" ++
       from.s ++ "&destId=" ++ stopDayTime.stop.s};
```

Multiple definitions for stop, day and time. 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 *StopDayTime*. As a consequence, when a user gives a new definition for an existing function, the linearization will be changed to the new one. For instance, the `WorkStopDayTime` function will be the following when the user defines a new meaning for work.

— U: work means Chalmers Tvärgata on Monday at 8:00

```
WorkStopDayTime = toStopDayTime TravelEng.Work
    TravelEng.St_1590 TravelEng.Monday "8:00"
```

3.2 Self-adaptation

Vehicle labels in our dialogue system are represented by type of the vehicle (e.g. bus, tram) and a number, like bus number 10. Assuming the following query, the web service offers Grön express bus.

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

Since the "Grön express" 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 desired vehicle label must be added to the GF 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. So the parser tries to parse the vehicle label and if it succeeds, 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.

```
fun
  Lbl_1 : Label;

lin
  Lbl_1 = toLabel "GRÖN EXPRESS";
```

By adding new labels to the grammar, the PGF file is updated and this answer will be shown to the user:

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

4 Example: An Adaptable Transport Query System

We have implemented an adaptable transport query system by applying the adaptation technique to the base line system.

The following conversation shows 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 short 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)

Some of the supported definitions in the adaptive query system are 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`)

This text-based query system can be extended to a multimodal dialogue system [5]. In such system, the users can define a synonym for a stop name, which cannot be recognized by speech recognizer. After that, the user can refer to that place by using a usual word in later dialogues. Moreover, the user adaptations can be saved in a log file, which will be retrieved when the stop grammar changes.

5 Evaluation

Our criterion of evaluation was to assess the effects of user adaption on speech recognition. To do this task, we tested 120 random generated input queries of our transport query system by an ordinary speech recognizer. 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[4]. After collecting the outputs of the

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

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:

- 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:

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

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

To sum up, user adaptation affects the speech recognition process and results in a more reliable system. This is due to the elimination of foreign words and shorter dialogues.

6 Conclusion

In this work, we presented an adaptation technique for the GF-based dialogue systems, which can be used for both user adaptation and self-adaptation. The user adaptation is based on the idea of voice programming that allows users to reconfigure dialogues systems by using natural language commands. These commands are

interpreted by the query system and then stored as a rule in the system. User adaptation results in concise dialogues and accurate speech recognition, which will increase the users' satisfaction.

The baseline system that was used for implementation of the adaptation technique is a transport query system that communicates with a transport web service and presents accurate travel plans to users. In this system, a GF writer application generates a set of grammar modules for stops dynamically, which makes the system portable to other transport networks. Moreover, we applied the system's self-adaptation by adding new bus/tram lines to keep the system always updated.

Acknowledgment. The author would like to thank Aarne Ranta for his enthusiastic guidance and suggestions through this research and for his valuable comments about this manuscript. The special thanks also go to Grégoire Détrez, Krasimir Angelov, Ramona Enache and John Camilleri for their undeniable help and productive discussions during this work.

References

1. Lemon, O., Pietquin, O.: Introduction to Special Issue on Machine Learning for Adaptivity in Spoken Dialogue Systems. ACM Transactions on Speech and Language Processing 7(3) (2011)
2. 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, Sydney, Australia, pp. 57–60 (2006)
3. 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, Prague, June 29 (2007)
4. Bringert, B.: Speech recognition Grammar Compilation in Grammatical Framework. In: Proceedings of the ACL 2007 Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic, June 29, pp. 1–8. The Association for Computational Linguistics (2007)
5. Bringert, B., Cooper, R., Ljunglöf, P., Ranta, A.: Multimodal Dialouge System Grammars. In: Proceedings of DIALOR 2005, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France, pp. 53–60 (June 2005)
6. Bringert, B.: Embedded grammars. MSc Thesis, Department of Computing Science, Chalmers University of Technology (2004)
7. Georgila, K., Lemon, O.: Programming by Voice: enhancing adaptivity and robustness of spoken dialogue systems. In: BRANDIAL 2006, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue, pp. 199–200 (2006)
8. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011) ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth)
9. Angelov, K., Bringert, B., Ranta, A.: Pgf: A portable runtime format for type-theoretical grammars. Journal of Logic, Language and Information 19, 201–228 (2010)
10. Ranta, A.: The GF Resource Grammar Library. In: Linguistic Issues in Language Technology, LiLT, vol. 2(2) (December 2009)
11. Angelov, K.: Incremental Parsing with Parallel Multiple Context-Free Grammars. In: European Chapter of the Association for Computational Linguistics (2009)