Rapid development of application-oriented natural language interfaces

Tamás Mészáros
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Magyar tudósok krt. 2. I.E.437
H-1117 Budapest, Hungary
Email: meszaros@mit.bme.hu

Tadeusz Dobrowiecki
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Magyar tudósok krt. 2. I.E.437
H-1117 Budapest, Hungary
Email: dobrowiecki@mit.bme.hu

Abstract—**Computerized systems became everyday companions to humans in various application areas. Due to the increasing complexity of these systems and wide variety of problems they help solving the interface between humans and machines becomes an ever critical issue. Controlled natural languages might provide a good medium between humans and computers, however, they are not easy to design and maintain, and humans need time to adopt to them. Authors propose a solution to these problems by using a controlled user interface which is powered by an application-oriented natural language. The grammar rules and vocabulary of this language are automatically generated from a conceptual model of the user interface using so called extended conceptual graphs.**

I. Introduction

To provide an efficient and easy access to computerized systems is a complex task for computer science that became everyone's problem during the past decades. Creating interfaces between humans and information technology devices (computers, mobile phones, PDAs, even more advanced multimedia or household equipments) became a central problem of software (and hardware) developers.

The dawn of the World Wide Web widened the user community and the simplicity and understandability of user interfaces became key requirements. Conventional software systems (e.g. operating systems, desktop applications) adopted Web technologies in many ways to simplify and unify their user interfaces. With the dawn of the "Web is the desktop" slogan the means of building user interfaces are becoming uniform and common [1].

This is not the case with the content and services. Complex systems tend to have complex interfaces (sometimes users have to use complex interfaces to perform simple tasks, e.g. word processing in Office 2007). The end users' learning curve of these interfaces did not drop much enough by utilizing common techniques for creating them. Menus, buttons, forms and other interface elements are needed to provide the necessary access to services and information. No matter how clever their presentation is – the user has to select the right one and has to know where it is and how to use it.

Conventional user interfaces neither could handle well changes in the underlying software system well. Changing services, features, internal data types usually means related necessary changes in the interface. The adaptation from legacy systems is expensive for the customers and often does not yield a better solution for them. It is also of order to notice that users have aversion to changed interfaces. People with less training (or experience) in informatics usually seriously oppose that kind of changes.

The next problem with complex interfaces is that the required space to display all informative elements might not be available (e.g. in mobile phones). This problem is very hard to overcome with traditional techniques.

In this paper we propose an alternate way of building complex user interfaces: a method that relies on the expressiveness and universal understandability of natural languages. Although computerized natural language processing (NLP) has its problems and limitations (we will address them in this paper) they offer a natural way to cope with the complexity of user interfaces. They tackle the complexity at the language level and let us to keep computer interfaces relatively simple.

The problem of creating a natural language interface is twofold: analytic in understanding the user input (processing, understanding the meaning) and synthetic in providing results to the user (generation). In this paper we focus on natural language processing: understanding the user and providing an input interface with good expressive power is a more demanding task. Using natural languages for reporting and providing results to the user could not be desirable at all as computer visualization techniques might offer a better solution for this task.

Although this paper focuses on the task of natural language-based query interfaces the methods and tools presented here can be easily extended to other types of interfaces. They could be used in language generation for reporting and other applications.

The following chapters describe the construction of natural language interfaces in detail: the application and automatic generation of application-specific controlled languages and two prototype implementations.

II. Natural Language Interfaces

A. Natural Language Processing

Building complex user interfaces is not a new problem. Research in Artificial Intelligence was focused on developing techniques to understand and to use natural languages in computerized systems for a long time. Several methods were developed to parse human languages and to understand the meaning of texts written in natural languages. The task, however, is very complex and difficult to solve.

Despite the inevitable interest and serious efforts general language processing and understanding systems have not reached the level of everyday applicability. The main problem lies in the spatial and temporal diversity of natural languages, and the huge amount of necessary contextual knowledge to understand them.

B. Controlled natural languages

In order to use present NLP techniques in computerized systems some authors suggested “restrictions” to natural languages. So called controlled natural languages [2] resemble to ordinary languages but have a strict (and restricted) set of language rules, vocabulary and unambiguous meaning, therefore they could be more easily processed by computers. The restrictions allow the successful application of controlled language NLP systems by avoiding the problem of disambiguation and uncertain grammar rules, and by explicitly linking the language to the contextual knowledge stored in the system.

Typical application areas include information systems, technical documentation, and machine translation. Providing database systems with easily configurable and understandable graphical interface was a key problem in the 90s and some researchers suggested the usage of natural languages for DB access [3]. There are also experimental systems that utilize NLP techniques on their interfaces, e.g. the natural language bus oracle [4] and for question answering in knowledge bases [5]. Application of controlled languages were also investigated in the field of machine translation [16].

Unfortunately, there were several problems with the use of controlled languages that prevented their widespread application.

First, the controlled language is not exactly the same as a natural language known by the user – the user has to learn it. This could take time, and often there is no time to train the user (or there is little interest in the user to learn the restricted rules and vocabulary). The user might “adapt” to the language with time, but it is usually required (or desired) that the user should be able to use it from the very beginning.

Secondly, these languages are not flexible enough, they could not adapt to the changes in the underlying software system (e.g. in data structures). These changes usually render the controlled language obsolete – it has to be adapted manually. Such task requires programmers with a unique training, and the adaptations are usually not easy to perform.

In the next two chapters we will provide methods to increase the adaptivity in both cases.

III. Controlled Language Interfaces

To circumvent the problem of user adaptivity we propose controlled user interfaces to use restricted languages. These interfaces do not allow the user to form sentences freely – they monitor the user input and automatically adapt it to the language. They also provide help for the user on how to use the restricted language by suggesting possible language constructs, words or expressions.

The simplest way of building a controlled language interface is the predictive text input.

A. Predictive text input

The predictive text input was originally developed for numeric keyboards [6]. During the text input the system analyzes the user input, corrects errors and provides suggestions. In its original form this method is based on a simple dictionary – in our case it utilizes the controlled language.

The predictive language input method continuously analyzes the text the user types in, it determines the set of possible sentences based on that input, and provides suggestions to continue the typing at the cursor position.

With this technology we can attain two goals. The text input is constructed following the language rules and the user gets immediate help in using the controlled language. This way the user can use the interface from the beginning (it might be slow for the first time, yet it will be usable), and with time the interface language will become more and more familiar and easy to use.

This proposed technique raises some additional requirements for natural language processing. It should support processing sentence fragments and the generation of “continuations” (feature extensions). Furthermore, language processing and generation techniques applied in these systems should be efficient: they have to analyze the user input and provide suggestions in real time while the user is typing.

Chapter IV will present the details of effective language representation and how restricted languages are used in controlled user interfaces.

B. Architecture of a Controlled Language Interfaces

Fig 1 shows the proposed architecture of a controlled natural language interface that monitors the user input and provides suggestions based on the controlled language.

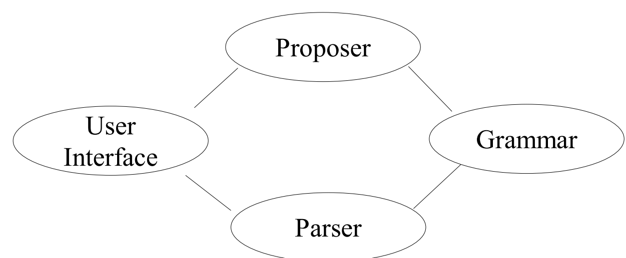

Fig 1: Architecture for controlled language interfaces

The role of the *User Interface* is to provide the text input method that continuously monitors the user input and displays suggestions to the user. It sends the actual state of the input to the *Proposer*, and makes use of the suggestions computed on this basis.

The task of the *Proposer* is to analyze the user input and provide suggestions. It parses the input text (sentence fragment), and determines the set of possible language symbols that can be added to the input according to the rules of the language. The elements of this set are returned to the *User Interface* as suggestions.

The user continues formulating the query sentence by successive typing in and choosing from the available suggestions. When ready, the completed query is passed to the *Parser*. Its task is to provide a complete analysis of the user input.

The *Grammar* component supports both the *Parser* and the *Proposer* by providing an efficient grammar representation and functions to access it.

The operation of the interface could be divided into two phases: generating suggestions based on the monitored user input and parsing the complete input. The first phase involves the *User Interface,* the *Proposer*, and the *Grammar*. The user input is continuously analyzed and suggestions are generated. When the user sends the complete request to the system (second phase), it is analyzed and the complete parse tree is sent to the backend to process it and handle the request (*User Interface, Parser*, and *Grammar*). Both phases use the same grammar representation but to different ends.

In the following chapter, we will closely follow on how this grammar is represented and used in the system. We will also address the problem of efficient grammar representation mentioned earlier.

IV. Representing and Using the Controlled Language

A. Grammar representation

In the proposed system the grammar is used in two places: for generating suggestions and for parsing user input. The selected grammar representation shall support both applications. The representation shall also be efficient when creating suggestions as this is required by the real-time nature of the user interface. In order to fulfill these requirements we have selected the simplest grammar type: the context-free grammar (CFG) [7]. From the point of view of our problem this grammar has some favorable features: its parsers are effective and easy to implement, and its expressiveness proved to be sufficient in our applications.

We applied some minor restrictions to CFGs in order to enhance the effectiveness of the *Proposer* and to ease the implementation of the first prototype. These restrictions are not obligatory.

In order to provide suggestions in the right order alternate symbol substitutions should be enumerated from the simplest to the most complex order. The simplest substitution is a terminal symbol, more complex is a list of terminal symbols, then non-terminals and their lists follow. The reason behind this design pattern is that this way the *Proposer* will generate the simplest suggestions first and no suggestion ordering algorithm is needed.

The second restriction was to eliminate recursive grammar rules (a symbol may not be substituted by itself or any other symbol that can by substituted by the original one). This simplified the implementation of the *Grammar* component.

Efficient parsers use the **packed forest** representation to store the CFG internally [7]. When using this representation several parse trees (a parse forest) could be represented using a single graph. Its nodes could represent symbols and also a set of symbols. This "compression" makes possible to represent exponential number of parse trees in polynomial space and time – an important feature when the aim is an efficient parser.

Generating suggestions is also an exponentially complex task, therefore we can use the same "packing" technique for efficient generation. The task of the *Proposer* can be described therefore as follows. First it finds parse trees (subset of the parse forest) that match the current input (i.e. sentence fragment). Secondly it attempts to extend the fragment to a complete sentence following the selected rules (trees). During this step a list of symbols (possible extensions) is generated. After removing duplicates and proper ordering the elements of this list are returned as suggestions.

In order to implement the *Proposer* we used a graph similar to the packed forest grammar representation called **ordered packed forest (OPF)**. It uses the same compression technique: it integrates parse trees into a common graph, but performs this differently. This graph is a directed bipartite graph which has two node types: symbols and symbol lists. Edges from symbol nodes to symbol list nodes describe symbol substitutions, and edges from symbol list to symbol nodes describe containments.

Edges starting at symbol nodes are XOR logically related: a parse tree can not contain more than one of them. This is the place where multiple parse trees are joined into the parse forest (compression). Edges pointing from symbol list nodes to symbol nodes are AND logically related: they are both needed in a parse tree to represent the grammar rule.

All edges are ordered and enumerated in same the order as the represented substitutions and symbol lists are ordered in the grammar. For symbol lists this is necessary to represent the grammar. For the list of substitutions this ordering ensures that suggestions will be presented in the proper order.

The selected grammar class (restricted CFG) allows to represent all symbols in the grammar with exactly one node in the OPF graph. This greatly reduces the complexity and facilitates the creation of suggestions.

With such grammar representation the task of the *Proposer* can be described as a graph traversal problem: we have to find a symbol node in the OPF graph which the user wants to extend, and then the tree should be traversed from

that node to the nearest terminal symbol nodes that could be added to the given point of the input. These terminal symbols will be presented as suggestions to the user.

Fig 2 shows as an example a part of an OPF graph. This is a part of a grammar used in one of our prototype implementations described in Chapter VII. Section A. It shows the rules for noun adjectives (VJ). Nonterminal symbols are represented with empty circles; terminal symbols are black circles; symbol lists are empty squares. All of three non-terminals (VJ, VJ1 and VJ2) have multiple possible substitutions (multiple XOR edges originated in the non-terminals). The related grammar rule prescribes substitutions from the simplest to the most complex order, as the edges originated from the nodes are ordered from left to right. All the substitution orderings of the grammar rule are retained in the graph representation by the suitable ordering of the edges.

B. *Large, dynamic vocabularies*

The previously described grammar represents the complete language in a single structure. This is not always desirable or even possible. There are cases where the complete vocabulary is too large or could not be enumerated in advance (e.g. it is changing dynamically, it contains too large number of terminal symbols, etc.). In such cases grammar representation would be too large and it would considerably slow down the operation.

The representation tackles this problem with the introduction of a special node: a terminal symbol with a generation rule, called Expandable Terminal Node. Such nodes do not contain a terminal symbol, instead they are equipped with a rule how to generate symbols. (Strictly speaking this node type is a non-terminal symbol that could be replaced by a single terminal symbol. These non-terminals are not covered by the grammar itself, the generation rule could be any algorithm that returns terminal symbols.)

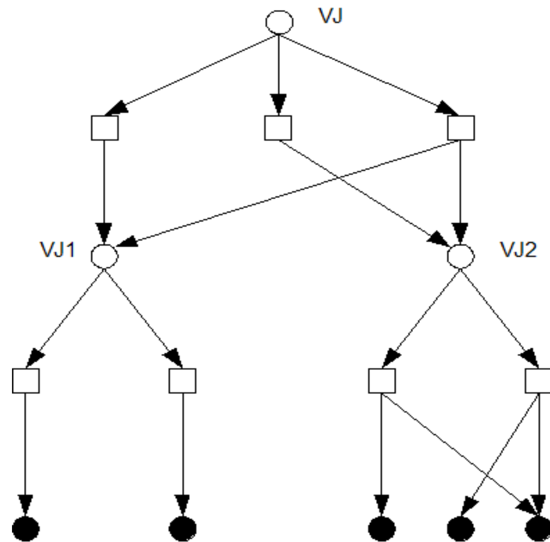

Fig 2: An example of the ordered packed forest representation

Whenever the suggestion generator reaches an expandable node it executes the expansion rule to generate the list of possible terminal symbols. During the parsing these rules provide a test method to check whether a symbol found in the input matches a terminal symbol generated by the expandable node.

An example of usage could be a list of language names covered by a general LANG terminal symbol. The generation rule could be e.g. a database query that returns the actual list of languages or tests an input for a language name.

C. *Generating suggestions with the OPF grammar*

The task is to provide terminal symbol suggestions to the user. In general, this is possible at any point of the user input, but for the purpose of this paper we focus on listing symbols that could follow the actual sentence fragments (extending the sentence at the end).

Enumerating the possible continuations consists of parsing and generation steps. Parsing determines the set of possible parse (sub)trees. Then a list of terminal symbols that are allowed after the last symbol of the input sentence are generated by the selected parse trees. In the selected grammar representation these steps are performed in a closely related manner using the same data structures. We have chosen a bottom-up parser algorithm to perform the parsing step, and a top-down generation algorithm to produce the suggestions.

The detailed description of the *Proposer*'s algorithm is out of the scope of this paper – only a short overview is presented here.

Step 1: the algorithm starts with identifying the terminal symbols already written by the user. This could be easily performed by a simple search in the array of the terminal symbol nodes. (Remember: each symbol is represented only once in the graph.)

Step 2: Next those symbol list nodes are found that contain the identified terminal symbols. Since the suggestions are generated only at the end of the sentence, the algorithm considers only symbol lists containing the last terminal symbol.

Step 3: Lastly these lists are analyzed. If one of these lists is incomplete at its end then a suggestion is generated that will contain the next terminal symbol. If there is a complete list then its parent non-terminal symbol is examined in a similar way than the found terminal symbols (Step 2).

There is a special case in Step 3 that can be handled separately to enhance the *Proposer*'s behavior. If the incomplete symbol list contains only terminal symbols the algorithm will return all missing terminals at once. It will propose complete phrases instead of single symbols thus providing a faster operation and better understandability for the user.

The proposed algorithm requires that the user follows the rules of the grammar. In order to assure it the controlled interface will reject any word typed in by the user that is not allowed by the *Proposer*.

D. *Parsing with ordered packed forest grammar*

The parser is based on the same bottom-up parsing method as the generator in its parsing phase.

Parsing starts with identifying the terminal symbol nodes then it attempts to traverse the ordered packet forest upwards to reach its root, the sentence symbol (S).

During the traversal the algorithm analyzes the edges with XOR relation and selects the rightmost edge (symbol substitution) from among the alternatives. These decisions will determine which parse tree is selected from the parse forest.

Ideally the controlled interface ensures that the complete parse tree can be reached. In practice, however, the algorithm should cope with incomplete user inputs (S could not be reached), and provide detailed error message to the user in such cases.

E. *Implementing the controlled interface*

Considering the proposed architecture (Fig 1), the complexity of the components, and the user interface requirements, the controlled interface could be implemented as a standalone or as a client-server application as well.

Problems with standalone applications (installation, maintenance, resource requirements) and trends in user interface developments mentioned in the introduction suggest that a Web-based implementation should be predefined.

During the design of a client-server Web-based implementation the capabilities of Web components and their resource limits should be closely considered.

At client side, modern Web browsers support the execution of program code of several forms, typically Javascript, Java, and Flash programs. Javascript is the most common, it is supported (and enabled) by most browsers by default. The other two options requires other software to be installed. All three common programming techniques allow dynamic data transfers between the client and the server.

The resource limitations in browsers suggest that only the *User Interface* component should be deployed there – the other components should be working at server's side. By choosing Javascript we ensure the greatest possible usability: all modern Web browsers support this programming technique.

The Javascript implementation in browsers supports dynamic client-server data exchange in the background using the AJAX (Asynchronous JavaScript and XML) technique [8]. This – along with the possibility to monitor and intercept user input – is sufficient to implement the *User Interface* component.

The other components can be implemented at server side where the resources and programming capabilities are vast – they practically do not limit the implementation.

In application environments where the described client-server implementation is not possible (e.g. systems with no networking capabilities) a standalone application can be considered.

In certain environments it is also possible to combine these methods. For example, mobile devices with limited (and costly) networking could use a system which deploys the *User Interface*, *Proposer* and *Grammar* components at client side, and the *Parser* along with a copy of the *Grammar* is present at server side. It is also possible that the *Grammar* at client side is not complete. It contains that part of the packed forest (graph) representation which is needed at a given time of the operation, and other parts are loaded dynamically as needed. This hybrid operation could avoid the installation and maintenance problems of a standalone applications and could work also with limited networking capabilities.

It is common for all implementation methods that the application-specific controlled language is separated from the interface program, thus the interface implementation is application-independent. This makes possible to create a uniform interface (or a handful set of interface implementations) for all application needs.

V. Automatically Constructing The Controlled Language

The previous chapters detailed how a language could be represented and used in a controlled interface to provide natural language text input for the user. The User Interface and the Proposer help in learning and following the rules of the controlled language. In the following, we will address the other problem of the application of controlled languages: weak adaptivity and difficult design.

In this chapter we will provide a method to facilitate the creation and maintenance of application-specific controlled languages. The idea is to automatically generate the language from an application-specific conceptual model of the user interface.

This model-based language generation is similar to model-based software development in a sense that they are both based on a domain model that is used (along with a set of pre-made application components) to create the full application (the controlled language interface in our case).

A. *Requirements for the Model*

Our goal is to provide a modeling framework that makes the design of controlled natural language user interfaces relatively easy. Programmers without any knowledge of natural language processing techniques and tools should be able to create and maintain controlled language user interfaces. Language grammars and vocabularies should be automatically generated from the model of the user interface.

Similarly to software modeling frameworks, a visual modeling system is needed, that has predefined, extensible, customizable building blocks, and that is capable of combining these blocks into an appropriate model of the user interface.

This model should also provide means to describe the relation between the interface and the application itself, the bridge to data sources (database, XML, software services, etc.) and to the services of the application.

It might be also desirable that the selected modeling technique could describe the interface (and the application concepts) in more detail making it possible to use artificial intelligence reasoning and traditional natural language processing techniques based on the model. This requirement is currently out of the scope of our research but it might provide additional benefits for future applications.

B. Domain modeling Using Extended Conceptual Graphs

After having analyzed the above requirements we selected the Conceptual Graph (CG) modeling technique [9]. It is a very flexible tool to define concepts and their relations. On one hand it allows the creation of loosely defined concept models, on the other hand it also makes it possible to construct detailed models that can support logic-based reasoning. It has an easy-to-understand visual representation, and it has some nice graphical editors [10].

CGs have been already applied in similar applications [11], and they were also proposed as a natural language grammar representation called Conceptual Graph Grammar [12]. The CG model representation is also an ISO standard [13].

CGs are capable of representing the interface concepts and relations in an easy to understand way but they lack some features that are required to generate the desired controlled language interfaces. Therefore, we have extended this modeling system with methods to describe two other levels of the same model: the data and the language. These modifications do not change the basic behavior of the CGs but extend the applicability of the model.

The new data level introduces bindings to the application. At this level concepts (and concept types) defined in the CG model could specify relations to the data sources in the application. These application bindings make it possible to dynamically create concept instances during the operation of the NL interface.

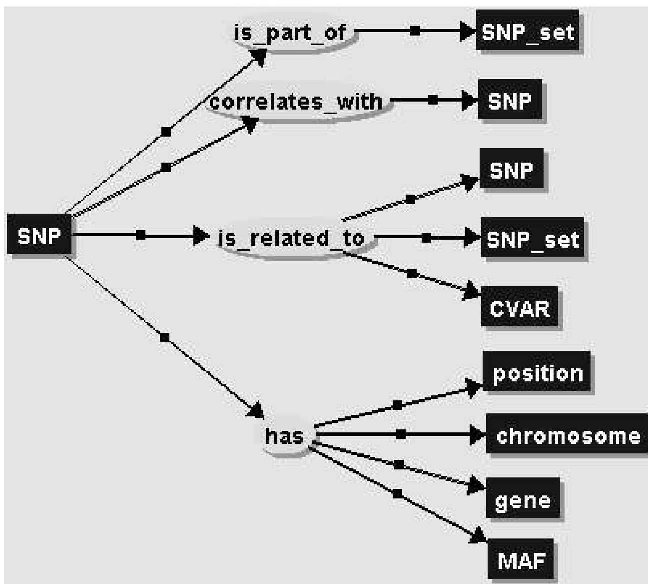

Fig 3: concepts and relations of a gene database interface (excerpt)

The language level adds special attributes to concepts and relations. The CG model is language-independent. Concepts and relations could be named using any notation and in any language. Our language extensions introduce language-specific constructs and symbols. They specify how a given relation or concept is represented in a given language. Language attributes of relations identify grammar constructs, while concept attributes define symbols in the context of languages and relations.

In order to store these extensions in the model we applied the GraphML standard [14] to describe the conceptual graphs. Although the CGs possess several representations (OpenCG, CGIF) these formats do not allow such extensions. GraphML is a standard way of representing graphs in a text file. It supports extensions and several graphical editors support this format.

Fig 3 shows an example conceptual graph of a genome application. The code fragment in Fig 4 shows a part of the extended CG of this model represented in GraphML.

C. Generating the Controlled Language grammar

From the model we can generate the controlled language in two steps: first relations (and concepts) form the basis of the language, the grammar, then the vocabulary could be constructed from concepts and their data bindings.

In order to create grammar constructs all relations of the model are enumerated. Grammar rules are formed from the language attributes of the relations and the attached concepts. As it was mentioned in the introduction we are focusing on the task of query interfaces in this paper. This means that these language attributes define query structures. E.g. from the "is_part_of" relation the algorithm generates lan-

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml ... >
  <!-- graph definition -->
  <graph id="GeneGraph01" edgedefault="directed">
    <desc>Extended conceptual graph for genome 
db</desc>
    <node id="101">
      <desc>SNIP: x</desc>
      <data key="type">concept</data>
      <data key="cname">SNP</data>
      <data key="instance">...</data>
    </node>
    <node id="102">
      <desc>relation between SNIPs</desc>
      <data key="type">relation</data>
      <data key="cname">is_related_to</data>
      <data key="lang_rule">...</data>
    </node>
    <node id="103">
      <desc>SNIP: y</desc>
      <data key="type">concept</data>
      <data key="cname">SNP</data>
      <data key="instance">...</data>
    </node>
    <edge source="101" target="102"></edge>
    <edge source="102" target="103"></edge>
    ...
  </graph>

</graphml>
```

Fig 4: GraphML representation of a conceptual graph (excerpt)

guage rules for describing a query for the part-of relation: “which SNP is part of an SNP set ...”. In addition to the query structures it also generates language rules for filtering structures e.g. “... an SNP which is part of an SNP set ...”.

D. Generating the Controlled Language vocabulary

The second step is to generate language symbols from concepts and relations. Concept and relation symbols from their language attributes are added to the language.

Concept instances (generated using concept data bindings) could also be added to the vocabulary. There is, however, no need to generate the complete vocabulary as it is mentioned in Section IV. B. The operation of the controlled language interface makes possible to store the instantiation rules (data bindings) in the grammar thus significantly reducing the size of the grammar. This way only application bindings are transferred from the conceptual model to the grammar (as special, expandable terminal symbols). These bindings will be activated (and transformed into real symbols) only when needed during the operation of the grammar.

VI. Summarizing the Interface Development Workflow

We can summarize the development of the application-oriented controlled language interface in the following steps

Step 1. User Interface Requirement Analysis

Analyzing necessary user interface functions (e.g. possible query and report structures) based on user and usage modeling. The goal is to identify concepts and their relations.

Step 2. Concept Modeling

Creating types and concepts based on the analysis, determining bindings between concepts and the application.

Step 3. Relationship Modeling

Specifying relationships between concepts that may be used (queried) in the interface.

Fig 5: Small part of the old Web form for the Hungarian Noun Database translated into English

Step 4. Creating the Conceptual Graph Model

Representing types, concepts, relations, and bindings to the application in a conceptual graph model.

Step 5. Language Extensions to the Model

Selecting interface languages (locales), assigning language structures to relations, extending concepts with words from the selected locales, customizing words according to the relations.

Step 6. Grammar Generation

Automatically generating the controlled language grammar from the model.

Step 7. User Interface Assembly

Selecting the appropriate interface implementation technique, and specifying (loading) the generated grammar.

Step 8. Validation

Testing and validation of the interface functions.

VII. Prototype Implementations

Two prototype applications were designed to evaluate our ideas in real-world scenarios. The first prototype was a Web-based query interface to the database of Hungarian nouns and their attributes. The second prototype was created to handle complex queries on a genome database.

A. Controlled Natural Language Interface to a Database

The Hungarian Noun Database [15] contains data about more than 30 000 nouns. For each noun it describes approximately 30 arguments and their relations. Typically users are looking for nouns with a certain set of arguments and with some specific relations. The traditional user interface to this database was a Web application. Noun arguments could be selected and relations could be described using a rather complex Web form (Fig 5). It contained a check box for all 30 arguments (only four are shown on the figure) and other fields for their relations. In some cases this form had to be used several times in a sequence to form the final query. In practice, users had difficulties handling the form in a single step, and more complex queries were out of their reach.

We replaced the traditional query interface with a simple text input (Fig 6) that used predictive text input and a controlled language to help the users in forming their queries.

Fig 6: Part of the new Web interface for the Hungarian Noun Database translated into English

The language was built by hand and it contained 10 rules and less than a hundred symbols.

We extended the original Web-based application with a client-side Javascript program using AJAX [8] for predictive text input and with a server-side code using our previously described parsing and generation algorithms to serve the client program with suggestions and to parser and translate the query into SQL.

The users found the new interface much easier to use, and they quickly understood how to form complex queries. They found the predictive text input technique hard to use for the first time, but they quickly adopted to it. Their main concern was that the query language was too strict and it did not allow certain query structures.

B. *Natural Language Interface to a Genome Database*

Our second prototype application is built for a rather complex database of medical data. In this database several tables contain information about genome and symptom profiles of patients. These tables are heavily inter-linked. The main application problem is that even expert users find very hard to form the proper SQL query structures to perform a search for a given class of patients, genome data, etc. The current interface contains predefined queries created by the developers of the application. These queries can be parametrized but the users are not allowed to change their structures or to form new kind of queries. Fig 3 in Chapter V. Section B. showed some concepts and their relations found in the database.

In order to allow regular users to form queries we applied our controlled natural language interface. First, the concepts and relations were modeled using techniques presented in this paper. Secondly, we generated the language rules and vocabulary from the model. This language was loaded into the Web system developed for our first prototype for testing, and we also implemented a standalone application in Java that used the same grammar and vocabulary.

Currently we are in the phase of testing the generated language and the Web and Java systems using real data and the previously defined queries.

C. Conclusion and Further Work

We proposed an approach to effectively use controlled natural languages on human-machine interfaces to query databases and software applications. The approach is based on two ideas: a controlled text input method and an automatically generated application-specific controlled language.

We developed a grammar representation that can be effective used for generating suggestions on the predictive input interface and for parsing user input. To circumvent the problem of authoring and maintaining an appropriate controlled language for the interface we propose the automatic generation of application-specific languages from a conceptual model of the application.

The proposed approach and algorithm has two principal advantages. First, it allows novice users to use the natural language interface from the beginning by providing suggestions. Secondly, it facilitates the automatic generation and maintenance of the applied controlled language. These advantages contribute to the successful application of natural languages in human-computer interfaces.

Two prototype applications were built. The controlled natural language interface to a noun and argument databased proved to be very useful and easy to understand for the users. Using a natural language interface for the genome database it could be possible for the users to build very complex queries that were not possible by traditional means.

Our research now focuses on the detailed elaboration of the conceptual modeling technique and its extensions in order to allow the automatic generation of more complex controlled languages. We also experiment with the application of our ideas in embedded systems (mobile phones, household equipments).

References

[1] A. Weiss, *WebOS: say goodbye to desktop applications*, netWorker, Volume 9 , Issue 4, pp. 18-26, 2005.

[2] Allen, Jeffrey; Barthle, Kathleen: Introductory overview of Controlled Languages. Society for Technical Communication meeting of the Paris, France chapter. 2 April 2004.

[3] I. Androutsopoulos, *Natural language interfaces to databases -- an introduction*, Journal of Natural Language Engineering, 1995

[4] T. Amble, *BusTUC: a natural language bus route oracle*, Proceedings of the sixth conference on Applied natural language processing, Seattle, Washington, pp: 1 – 6, 2000

[5] P. Clark et. al., *Capturing and Answering Questions Posed to a Knowledge-Based System*, Proceedings of the 4th international conference on Knowledge capture, Whistler, BC, Canada, pp: 63 – 70, 2007.

[6] S.L. Smith, C.N. Goodwin, *Alphabetic Data Entry Via the Touch-Tone Pad: A Comment*, The Mitre Corporation, HUMAN FACTORS, 13(2) pp 189-190., 1971.

[7] J. Earley, *An efficient context-free parsing algorithm*, Communications of the ACM, Volume 13, Issue 2, pp. 94-102, 1970.

[8] J.J. Garret, *Ajax: A New Approach to Web Application*, Adaptive Path,http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005. Retrieved on 2008-05-20.

[9] S. Polovina, *Conceptual Graphs: An Overview, Software and Issues*, A nearplanet.com White Paper, Retrieved on 2008-05-20.

[10] *CharGer: A Conceptual Graph Editor*, http://charger.sourceforge.net/, Retrieved on 2008-05-20.

[11] J.F. Sowa, *Conceptual Graphs for a Data Base Interface,* IBM Journal of Research and Development 20(4), 336–357, July 1976

[12] S.B. Johnson, *Conceptual Graph Grammar – A Simple Formalism for Sublanguage*, Methods Inf Med. 37 (4-5): pp. 345-52. 1998

[13] Information technology — Common Logic (CL): a framework for a family of logicbased languages, International Standard, ISO/IEC 24707, 2007

[14] GraphML, http://graphml.graphdrawing.org/, Retrieved on 2008-05-20.

[15] M. Kiss, *Főnévi vonzatosság a magyar nyelvben (Noun arguments in the Hungarian language)"*, Ph.D. dissertation, Eötvös Lóránt University, 2005.

[16] E.H. Nyberg and T. Mitamura, *Controlled Language and Knowledge-Based Machine Translation: Principles and Practice*, Proceedings of the First International Workshop on Controlled Language Applications (CLAW), pp. 74—83., 1996.