A Compiler-Interpreter-System for Decoding the User's Intention Within a Speech Understanding Application

Michael Ebersberger, Johannes Müller, Holger Stahl

Institute for Human-Machine-Communication, Munich University of Technology
Arcisstraße 21, D-80290 Munich
email: {ebe,mue,sta}@mmk.e-technik.tu-muenchen.de

Abstract: For a speech understanding graphic editor, a compiler-interpreter-system is introduced to process a semantic structure, a special form for representing the semantic content of a spoken utterance. After a semantic structure has been converted to database queries by the compiler, the interpreter processes all these queries and updates the corresponding database, i.e. in our case the graphics data base containing the features of all objects on the screen. The system has been tested with 1843 semantic structures of spoken utterances within the 'graphic editor' domain. The rate for correct database queries is 97.3%. Storage of the whole domain-specific knowledge in external and editable files enables easy portability to other domains.

1 Introduction

To demonstrate a speech understanding system, we implemented a speech understanding graphic editor, which allows the user to create, modify or delete three-dimensional objects such as cones, cuboids, cylinders or spheres only by spoken commands. The system reacts upon a spoken input by a graphic output on the screen and synthetic speech. The speech signal is recorded and preprocessed to an observation sequence *O*.

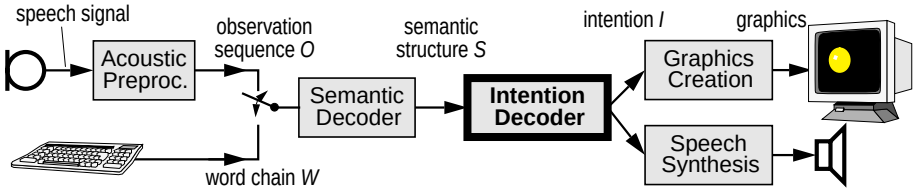


Fig. 1: Block diagram of the application ‚speech understanding graphic editor'

By maximizing the conditional probability $P(S|O)$ [7][8], the semantic decoder computes the most probable semantic structure *S*, which is used to find the user's intention *I*. Subsequently, operations on a graphic database are performed. The data of all objects shown on the screen are saved in this database and used to generate a pixel sequence, which is displayed on the screen. Furthermore, the database provides the possibility to pass textual information, which is given out by synthetic speech.

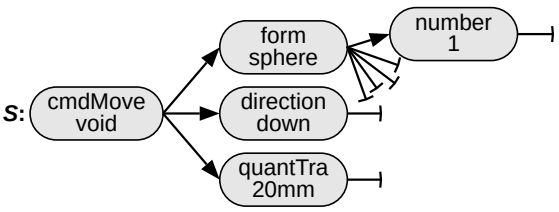


Fig. 2: Semantic structure *S* of an utterance

The semantic structure as semantic representation of a spoken or written utterance forms the input of the intention decoder. The user‘s utterance *"move the sphere two centimetres downwards"* is transformed by the semantic decoder into the semantic structure *S*, which is depicted in figure 2. In general, a semantic structure has the following characteristics [3]:

- The semantic structure *S* is a tree consisting of a finite number *N* of semantic units (simply called *semuns*) s_n: $S = \{s_1, s_2, \ldots, s_n, \ldots, s_N\}$
- Each semun s_n has a type $t[s_n]$ and a value $v[s_n]$. It can be drawn as $\begin{matrix} t\,[s_n] \\ v\,[s_n] \end{matrix}$.
- Each semun s_n refers to a certain number $X \geq 1$ (depending on $t[s_n]$) of successor semuns $q_1[s_n], \ldots, q_X[s_n] \in \{s_2, \ldots, s_N, \text{blk}\} \setminus \{s_n\}$, connected by edges "⟶".
- The blank-semun ’blk’ forms an exception. It represents a leaf of the tree, drawn as "⟙". It has the type $t[\text{blk}] = \text{blk}$, no value and no successor.

2 General Approach for the Intention Decoder

For decoding the user’s intention, i.e. the execution of the desired actions, a combination of compiler and interpreter is suggested. The preprocessed semantic structure forms the source language P_Q for the compiler. It represents the intention of the user to manipulate graphic objects. The result of the compiler is a program in the intermediate language P_I, which has especially been designed for that purpose. This intermediate language provides commands for arithmetic operations, control of data flow and database operations. The interpreter executes the instructions of the intermediate language by loading object data from the database, modifying these data and storing data of new objects into the database.

Fig. 3: Block diagram of the intention decoder as compiler-interpreter-system

Before using this system, a system administrator describes each semun with commands of the intermediate language P_I. The semun and the commands represent the same information. Additional semuns can be easily integrated at a later stage. (The use of a database language like SQL [5] is possible, too. For that reason, this article describes neither the structure of the intermediate language nor the functioning of the interpreter).

3 Preprocessor

Before a semantic structure is executed by the compiler-interpreter-system, it is possibly simplified or split by a relational preprocessor [1]. To keep the intermediate language simple, to minimize the compilation time and to reduce the instruction complexity, the preprocessor can split a semantic structure at special places, which represent for example the combination of two attributes or actions. The semantic structure of *"create a sphere and a cube"* is divided into two semantic structures corresponding to *"create a sphere"* and *"create a cube"*. The semun representing the *"and"* was eliminated.

4 Compiler

The input program of the compiler (in source language P_Q) consists of one or more semantic structures. These semantic structures can only be transformed in database actions, if the context sensitivity of natural language has been eliminated.

4.1 Contextual Sensitivity

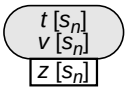
The following two word chains may introduce the problem of context sensitiveness.

- W_1: *"move a sphere above the cone to the right"*
  The system has to find a sphere in the database, which should be moved.
- W_2: *"paint a sphere"*
  The sphere is an attribute of a new object. The system must not search for that sphere.

In both cases, *"sphere"* is represented by the same semun. Only the context of the word gives some advice about the meaning. For that reason, the *status model* and the *status transition model* are established to describe the context of a semun.

4.2 Status Model

For each semun s_n, a status $z[s_n]$ is introduced. The number of all possible states is described in a status model. Within the 'graphic editor' domain, we defined five different states:



- The **cmd**-status is the command status of a semantic structure.
- The **new**-status collects attributes for a new object.
- The **what**-status searches for objects in the database.
- The **how**-status integrates new attributes for already existing objects.
- The **how-much**-status collects quantitative information.

4.3 Status Transition Model and Status Analysis

Each semun is understood as a status transition machine, which switches to a following status considering the type, the status and the successors of that semun. The sequence of states is described in the *status transition model* for all combinations of any type and any status. If the semun with the type "cmdMove" has got the status "cmd", its first successor-semun gets the status "what" (objects are searched by the system), its second successor-semun gets the status "how" (how are the objects manipulated), and the third successor gets the status "how-much" (which quantity of objects are manipulated).

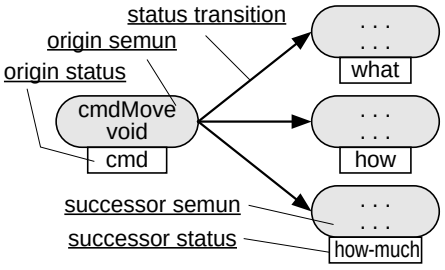


Fig. 4: Example for a status transition

During the status analysis, the status of each semun s_n within S has to be examined. For that purpose, the status transition model is used. The whole semantic structure is processed and the actual status for each semun is extracted. The actual semun status represents the actual context of this semun.

4.4 Production of Intermediate Code

The knowledge of each semun about its context is used to receive a new representation of the semantic structure. With the parameters type $t[s_n]$, value $v[s_n]$ and status $z[s_n]$, each semun s_n is transformed into certain commands, which are stored in a command model. In this model, each semun is linked with a series of commands, called basic commands, which can be processed by the interpreter. The semantic structure is transformed into a *basic command tree*, which has to be linearized before being processed by the interpreter. The execution of the commands modifies the actual knowledge of the system, which consists of a number of variables.

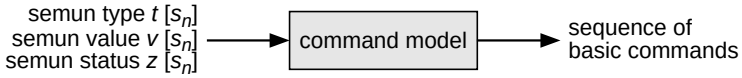


Fig. 5: Determination of the basic commands by the command model

4.5 System Knowledge

Whereas the compiler uses the system knowledge for common operations, the interpreter modifies the system knowledge by executing basic commands. The system knowledge is actualized until the whole basic command tree is processed. The system knowledge consists of two types of variables:

- **Internal variables** cannot be defined by the user. The system itself manages the handling of theses variables. For example, the result of searching operations is stored in internal variables.
- **External variables** can be defined by the user and are needed for the calculations. For example, the utterance *"move the cylinder to the right"* does not mention how far the cylinder has to be moved, thus the system uses external variables as defaults.

4.6 Linearizing the basic command tree by a "top-up" approach

The basic command tree has to be linearized for the interpreter. To determine the status $z[s_n]$ of each semun s_n, the basic instruction tree has to be processed "top-down" using the status transition model. However, commands can only be generated "bottom-up" [2], since basic commands in lower levels of the tree include some information, which is used in higher levels. Therefore a combination of both methods is used "top-up". Note in fig. 6 the three main steps 'DESCEND' for descending the semantic structure, 'CONSTRUCT' for constructing the linearized commands and 'FINISH' for terminating.

5 Results

The intention decoder has been tested with 1843 semantic structures of spoken utterances, which have been collected from 33 subjects during a Wizard-of-Oz simulation within the 'graphic editor' domain [4]. For the corresponding semantic structures, the rate for correct database queries amounts to 97.3%. It is impossible to reach 100%, since the execution of a language representation becomes more complex, the closer to natural language that representation is located [6]. In fact, the semantic structure claims to be close to the word level [3]. Due to the universality of the interpreter commands and the status analysis, as well as the storage of domain-specific knowledge in external files, the portability of the intention decoder to other domains is easily possible.

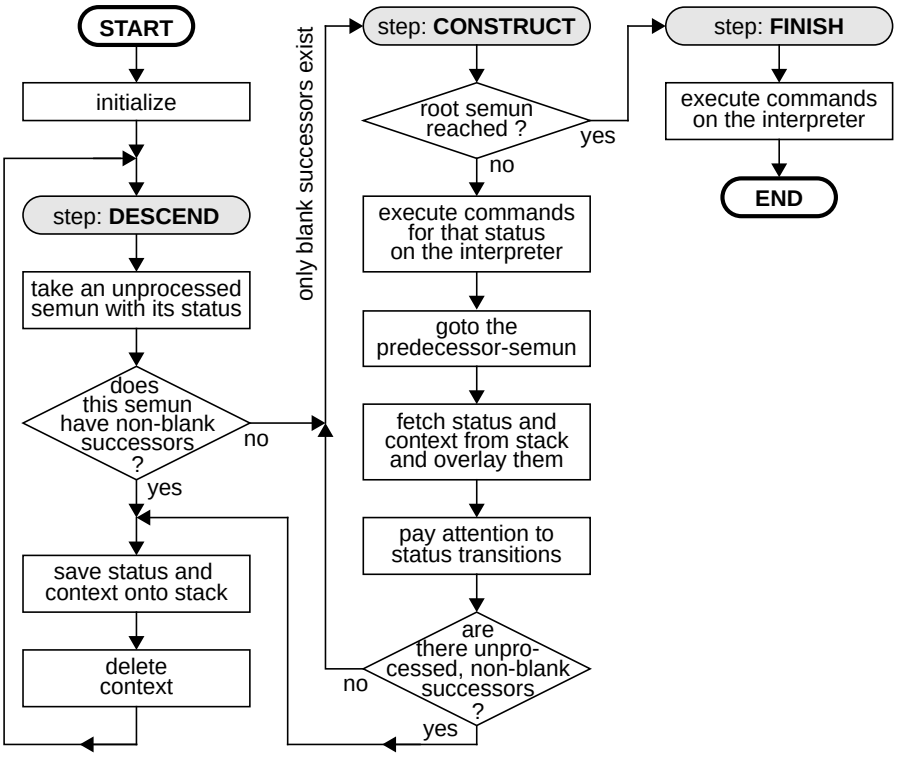


Fig. 6: Flowchart for linearizing the basic command tree

6 Acknowledgement

An online 'graphic editor' including the described intention decoder for German text input is running on WWW. If you try it, please be aware of out-of-vocabulary errors. The internet-address is: http://www.mmk.e-technik.tu-muenchen.de/~mue/nasgra/

References

[1] A. Aho, R. Sethi, J.D. Ullmann: *Compilerbau*, Addison Wesley, 1988

[2] L. Goldschlager, A. Lister: *Informatik: Eine moderne Einführung*, Hanser, 1990

[3] J. Müller, H. Stahl: *Die semantische Gliederung als adäquate semantische Repräsentationsebene für einen sprachgesteuerten ‚Grafikeditor'*, in L. Hitzenberger (ed.): Angewandte Computerlinguistik, „Sprache und Computer" (No. 15), Georg Olms, 1995, pp. 211-225

[4] J. Müller, H. Stahl: *Collecting and Analyzing Spoken Utterances for a Speech Controlled Application*, Proc. Eurospeech 1995 (Madrid, Spain), pp. 1437-1440

[5] D. Petkovic: *SQL die Datenbanksprache*, McGraw-Hill, 1990

[6] R. Pieraccini, E. Levin, E. Vidal: *Learning how to Understand Language*, Proc. Eurospeech 1993 (Berlin, Germany), pp. 1407-1412

[7] H. Stahl, J. Müller: *A Stochastic Grammar for Isolated Representation of Syntactic and Semantic Knowledge*, Proc. Eurospeech 1995 (Madrid, Spain), pp. 551-554

[8] H. Stahl, J. Müller, M. Lang: *An Efficient Top-Down Parsing Algorithm for Understanding Speech by Using Stochastic Syntactic and Semantic Models*, Proc. ICASSP 1996 (Atlanta, USA), pp. 397-400