at this moment. We hope that this will widen the applications of structural models and will improve the explanation facilities of expert diagnosis systems.

The ongoing research is carried out to implement the extended structural modelling schema. Two alternatives are checked, namely, the development of the expert system from scratch using one of the object oriented programming languages, and to integrate structural modelling system with one of the frame based expert system shells. In future the research and implementation of temporal reasoning methods are planned.

REFERENCES

[1.] Grundspenkis J., Causal Domain Model Driven Knowledge Acquisition for Expert Diagnosis System Development. Lecture Notes of the Nordic-Baltic Summer School (K. Wang and H. Pranevicius Eds.), Kaunas University of Technology Press, Kaunas,.Lithuania, 1997, pp. 251 - 268.

[2.] Grundspenkis J., Automation of Knowledge Base Development Using Model Supported Knowledge Acquisition. Proceedings of the Second International Baltic Workshop on Databases and Information Systems, Tallinn, June 12-14, 1996 (Hele-Mai Haav and Bernhard Thalheim Eds.), Vol.1, Institute of Cybernetics, Tallinn, Estonia, 1996, pp. 224 - 233.

[3.] Durkin J., Expert Systems. Design and Development, Macmillan Publishing Company, New York, 1994.

[4.] Even A.,. Temporal Logic. Nonclassical Logic (in Russian), Moscow, Nauka, 1970, pp. 124 - 190.

[5.] Kondrashina E., Litvinceva L,, Pospelov D., Representation of Knowledge about Time and Space in Intelligent Systems (in Russian). Moscow, Nauka, 1989.

[6.] Orchi T. Lecture Notes in Temporal and Deductive Databases, Royal Institute of Technology and Stockholm University, Sweden, 1995.

[7.] Prior A., Past, Present and Future, Oxford, Cleredon, 1967.

[8.] Tichy P. The Logic of Temporal Discourse. Linguistics and Philosophy, N.3,1980, pp. 343 - 369.

[9.] Vushkans Z. Development of Method for Addition of Temporal Dimension to the Functional Model, MSc. Thesis, Riga Technical University, 1997.

USING NATURAL LANGUAGE PROCESSING TOOLS FOR READING TEXTS IN A FOREIGN LANGUAGE

Tiit Roosmaa

Institute of Computer Science, University of Tartu

2 J. Liivi Tartu , EE2400 ESTONIA

Tiit.Roosmaa@ut.ee

Abstract

This paper describes results obtained within the EC Copernicus language technology project GLOSSER. GLOSSER aims to apply natural language processing techniques, especially morphological analysis, dictionary and corpora processing, to technology for computer-assisted language learning. Created by the GLOSSER team the linguistic components (Analyzer, bilingual dictionaries and bilingual corporas) and developed technologies - tested on three languages, can be used in a several information systems, where natural language analyze is needed.

Introduction

The GLOSSER prototype will help many people who know a bit of English but cannot read it quickly or with full understanding. It will support their knowledge of basic grammar (i.e., morphology) and remove the tedious task of thumbing through a dictionary.

GLOSSER's aim has been to implement bilingual dictionary (English-Estonian, English-Hungarian, English-Bulgarian) and bilingual corpora for the same languages as on-line context-sensitive translation support. It presupposes that a user has a text in his word-processor that he wants to read. Clicking on a sequence of words will display

a context dependent translation and or request, examples from the available corpora. The GLOSSER prototype carries out a morphological analysis of the sentence in which the selected word occurs and a stochastic disambiguation of the word class information. This information then matched against the dictionary and corpora. the human look-up process has been analyzed to design a user-friendly human-computer interface.

GLOSSER shall attempt to enable speakers of Bulgarian, Hungarian or Estonian, intermediate language learners and users of English, to read more fluently and make learning English easier. We imagine that they might read, i.e., a software manual on the screen. Upon encountering an unknown word or an unfamiliar use of a known word, for example, *reverts* as in:

This action ***reverts*** *the buffer to the form stored on disk.*

the user can mouse it to invoke online help (follow a dynamic hyperlink). Help will then provide the following facilities:

1. A morphological parse, showing *revert* as the stem of reverts,
2. The entry to the word revert in a bilingual English/X dictionary and/or a monolingual English one,
3. Access to similar examples of the word in online bi-lingual corpora.

GLOSSER applies natural language processing techniques, especially morphological analysis and disambiguation, dictionary and corpora processing, to technology for intelligent computer assisted language learning. There are four language pairs currently supported by GLOSSER: English-Estonian, English-Hungarian, English-Bulgarian and French-Dutch. Different partners have used different platforms for developing the GLOSSER prototype (UNIX, WINDOWS). The 32-bit Windows version (Windows NT and Windows 95) of GLOSSER is described in this paper. UNIX version of GLOSSER is described for example in (1).

Principal design of the prototype

GLOSSER architecture (figure 1) connects modules for morphological analysis and disambiguation, dictionary access and corpora search with an output module.

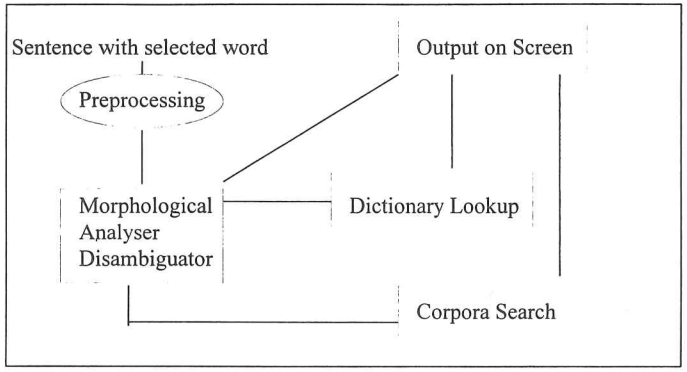

Figure 1: GLOSSER Architecture

We exploit available existing technology wherever possible, but substantial effort is dedicated to improving accuracy. For example, to make the morphological analysis begin from a most likely hypothesis about the part of speech, and to attempt to provide a most likely word sense for dictionary look up. A modest prototype demonstrating these capabilities can be built using well-researched rule-based technology, but it is certain that purely rule-based methods will discriminate too little in choosing relevant lemmata and word senses. To improve accuracy, we employ stochastic techniques for part-of-speech and other statistical methods for word-sense identification.

The rule-based Lexical Transducer and the stochastic Hidden Markov Model (HMM) for English with the associated software for morphological analysis and disambiguation have been developed by the Xerox Corp. for Macintosh machines. As one of the first concrete steps in the GLOSSER project, we ported these two systems to the 32-bit Windows platform. All the relevant parts of these systems have been re-written and compiled with Microsoft Visual C++ for Windows.

Morphological analysis and disambiguation

The disambiguator is more than a simple morphological analyzer. Both tools are developed by Rank Xerox (2) and ported to 32-bit Windows platform by

Morphologic. The GLOSSER prototype's lemmatizer provides the user with the information consisting of the actual wordform and the syntactic tags. A sample, run of the Xerox Lexical Transducer's Windows NT version used the lemmatizer of GLOSSER is presented here:

Input: This action reverts the buffer to the form stored on disk.

Output:

this+DT
action+NN
revert+VBZ
the+AT
buffer+NN | buffer+VB
to+IN | to+TO
the+AT
form+NN | form+VB
store+VBD | store+VBN
on+IN | on+JJ | on+RB
disk+NN

If words are ambiguous, then providing morphological analysis for them may be too tiresome to be of genuine use to language learners. The solution to this problem is disambiguation: to find the right entry in the dictionary, the HMM based part-of-speech (POS) disambiguator is applied before the morphological analysis in order to obtain the contextually most plausible morphological analysis.

Dictionary

The most suitable dictionary for the project (for different reasons) is the English Dictionary for Speakers of Estonian (3). It is a bilingual advanced learner's dictionary of a special semi-bilingual type: all explanations are given in English while at the same time all words and expressions have been translated into the native language of the user. The word-list contains about 30.000 words. The dictionary is supplied with a reversed index. It belongs to the series of Kernerman Semi-Bilingual Dictionaries. Kernerman's base is taken from the Chambers Concise Usage Dictionary (1985. Edinburgh: W&R Chambers), which is an English (advanced) learner's dictionary. It

has its equivalents in Hungary (English Dictionary for Speakers of Hungarian, 1992) and Bulgaria (English Dictionary for Speakers of Bulgarian, 1992). The structure of the dictionary consists of different information fields. We use the generalised markup language (SGML) for the structural analysis of the text.

The English/Hungarian dictionary window is shown in Figure 2. The number of hits is shown in the title bar of the Dictionary Window and the hits themselves are numbered in the left sub-window, for the sake of easy use.

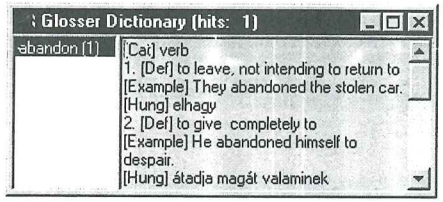

Figure 2. The dictionary window.

Corpus

A major source of information for users is the availability of examples of word use. To provide the learners with a sufficiently rich context, examples should fulfil the following requirements: 1. examples should come from different texts, 2. examples should include multiple forms, that is, inflections these exist, 3. translations should be provided if possible. The third and first requirements were fulfilled by providing parallel corpora of specially collected texts.

The *GLOSSER Corpus Window* is a child window in the GLOSSER main window. It consists of three parts: the hit (that is the result of the lookup according to the actual *Options*), the context of the actual occurrence, and its meaning in the target language. If there are more hits, they are listed one after another in the left part of the Window. If the user clicks on any of the elements of this list box, an adequate explanation and the text containing the input in question occurs in the upper right part of the window. The lower right part contains the translation of the upper right window's text .

The number of hits is shown in the title bar of the *Corpus Window* (Figure 3.) and the hits themselves are numbered in the left sub-window, as in case of the *Dictionary Window*.

Figure 3. The Corpus Window

User Interface

This GLOSSER prototype has a single source language, English and three target languages: Estonian, Hungarian and Bulgarian. With the help of the flags in the GLOSSER toolbar the user can set the language version X he needs. It means that both the dictionary window and the corpus window will show the adequate English-X language pairs.

The Input Window is a simple text window, which allows the user to type and edit a text according to the well-known Windows-standards. The content of Clipboard, however, appears in this window when starting the GLOSSER Prototype. This feature makes the direct communication between a word-processor (in our case, Microsoft Word for Windows 7.0) and the GLOSSER module possible. The user can start the full GLOSSER procedure by clicking the "G", that is, the *GLOSSER smart icon* in Word's toolbar as shown in Figure 4.

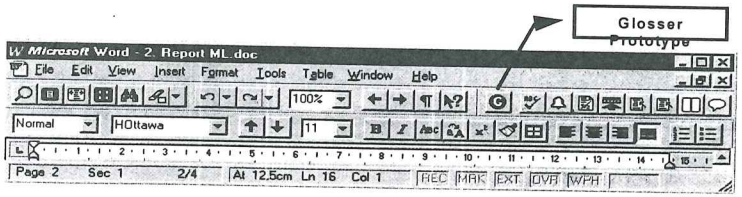

Figure 4. GLOSSER icon in MS Word's toolbar

With the help of smart icons the user can open new corpora, look-up for words in dictionary and corpus. When the GLOSSER window (see Figure 5) appears, the user can choose any part of the text shown in the Input Window. The marked text can be analyzed (by clicking on the Analyze button) or any word can be looked up (that will be shown in the dictionary and/or corpus, according to the actual options). If the *Analyze* button is pressed, the marked text will be analyzed morphologically word by word. Every possible segmentation of every word chosen will be shown in the *Disambiguator Window* by their lemmas and potential morpho-syntactic tags. Lemmatizing is done by the Xerox morphological analyzer's Win32 version, and disambiguation is done by Xerox's part-of-speech tagger's Win32 version introduced above.

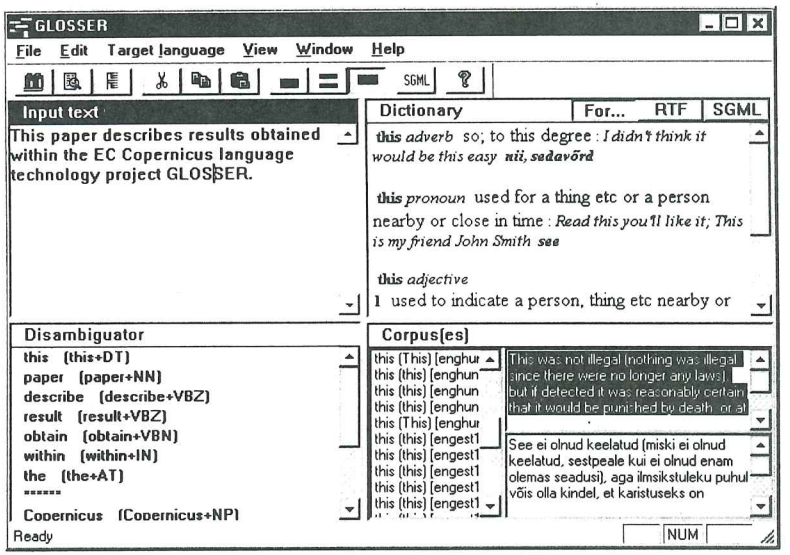

Figure 5. GLOSSER main Window

There are two more icons in the toolbar, one for the SGML layout and one for the Help.

Conclusions

The integration of existing language technology tools: morphological analysis and disambiguator, dictionary and corpus processing, has led to a powerful tool - GLOSSER prototype. Other languages could be easily implemented in the GLOSSER.

Acknowledgements

The Copernicus program of the European Commission supports the GLOSSER project in grant #343. The GLOSSER system has been developed by a team composed by Rank Xerox Research Centre in Grenoble (France), University of Groningen (The Netherlands), Linguistic Modelling Laboratory, Bulgarian Academy of Sciences (Bulgaria), Morphologic Ltd (Hungary), University of Tartu and Institute of Estonian Language (Estonia). I would like to thank for their contribution all the members of the GLOSSER team.

References

(1) Nerbonne, John and Petra Smit. GLOSSER-RuG in support of reading. In Proc. of COLING '96, pages 830-835, Copenhagen, 1996

(2) Bauer, Daniel, Frederique Segond and Annie Zaenen., LOCOLEX: Translation rolls off your tongue. In proceedings of the conference of ACH-ALLC '95, Santa Barbara, USA, 1995

(3) Password: English Dictionary for Speakers of Estonian. TEA, Tallinn, 1995

Meta-model based data conversion

Janis Plume, Juris Strods, Ivars Karlsons, Uldis Smilts, Juris Smotrovs, Guntis Gavars, Indra Stasko, Maris Dancis
Riga Information Technology Institute
Skanstes iela 13, LV-1013 Riga, Latvia
Phone: +371 7 821457
e-mail: janis.plume@dati.lv

Abstract.
The article proposes a method for development of data conversions. The method is based on meta-models and allows to separate logical data from physical representation of the data. The article describes a development framework, which supports the development of meta-model based data converters.

1. Introduction

Data exchange between different environments becomes more and more actual task nowadays. This is due to the enormous variety of the application programs, development and CASE environments that are developed up to now. There are various methods for data conversion among the different environments.

A method of data conversion is described in this paper. This method is based on so called meta-models and offers a highly flexible framework for development of data conversions. The method was initially meant for CASE data conversions but it proved to be very useful for various other areas, too.

In general under data conversion between environments A and B is assumed a conversion of data stored in a data exchange format of the environment A to a data format of the environment B. Term "environment" here means any software tool or application which has its data exchange format. In reality this term can be used much wider - under environment meaning any data format which has fixed syntax (e.g. programming language, CASE tool's import/export format etc.).

The second section contains description of data and meta-data methodology [1, 6].

The third section contains a short description of functioning of the meta-model based data converter [2,4,5].

The fourth section contains a description of the meta-model based data converter development methodology [3].

The fifth section focuses on the application areas of the data conversion method described here.

2. Data and meta-data

This section contains the basic concepts and description of the methodology of meta-models. There are various means for data abstraction which are used depending on information which has to be interpreted. For instance, formal grammars are used to describe formal syntax of programming languages, entity-relationship (ER) models are used to describe data base structure.

Meta-model approach is mainly based on the ER modelling and uses all basic concepts of the ER modelling.

Origins of the meta-model ideology come from the CASE tools and that is why further explanation of the ideology will be mainly focused on application of meta-models in this area.

In the meta-modelling a four level data architecture [1] is used to describe the data. Let us have a look at an example of the four levels of data.

Applications process information that is called *data* - the first level. For instance, "customer Smith places order NR. 12341" is a typical example of data.

Such data can be described using a *model* - the second level of the four level architecture. An example of the model in the previous example could be "customer places order" or using entity-relationship notation shown in the Figure 1.

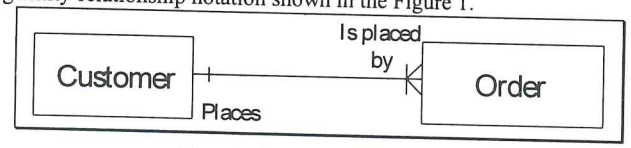

Figure 1. An example of ER model

CASE tools are used to store information models, be it the ER-model, data flow model, function decomposition model or any other. A question arises from this - what are the modelling means that are allowed for the CASE tool. What type of models can be developed using a CASE tool ? To answer such questions strictly, there can be used so called meta-models (the third level). *Meta-model* is an ER model which describes what type of objects are used for model development, what relationships are among them and what attributes the objects have. To describe a CASE tool, which allows creating the ER, diagrams like that in Figure 1, the meta-model shown in the Figure 2 can be used.

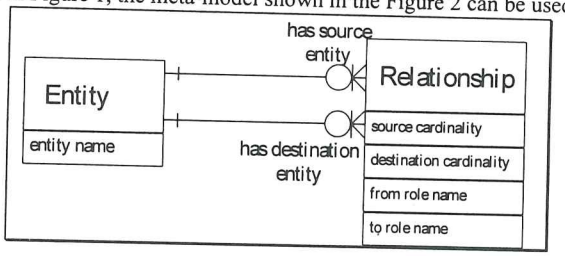

Figure 2. An example of meta-model

Let us have a closer look at a meta-model, because meta-models are essential components of the data conversion method described below. Symbols depicted by the

rectangular boxes are meta-model entities or meta-entities (Figure 2). Two meta-entities are present in the example - "Entity" and "Relationship". Those meta-entities are connected together using two meta-model relationships (meta-relationships) "has source entity" and "has destination entity". Meta-relationships define the source and the destination entities for relationships in a model. Meta-entities have their attributes (meta-attributes). The model in Figure 1 consists of the *instances* of the meta-model in Figure 2, namely, "Order" and "Customer" are instances of the meta-entity "Entity", relationship <places order, is placed by> is an instance of the meta-entity "Relationship".

Instances of meta-relationships are not so evident in the model - in this case the connection point of relationships with entities represents instances of meta-relationships ("has source entity" and "has destination entity"). Instances of the meta-attributes in the model appear in different forms. Instances of the meta-attributes "entity name", "from role name" and "to role name" appear as entity names ("Customer" and "Order") or role names ("Places", "Is placed by"). Meta-attributes "source cardinality" and "destination cardinality" appear as ends of relationship (+——K).

As we can see, the example of the meta-model defines the means for a simple entity-relationship diagram modelling. The meta-model for the ER diagram modelling can be improved (as it usually is in CASE tools) to include constructions like subtyping of entities, attributed relationships, n-ary relationships etc.

For each data, be it a data base, a program in a programming language or an import/export file of a CASE tool, there can be produced an ER-model (not unique) which describes the structure of the data. Using previously defined terminology, all data may be considered as a model and for each model a meta-model can be created which defines all possible models of a kind. Such approach will be used for conversion of data because the meta-model allows to operate with instances of logical objects (meta-objects).

The next thing is to define means used for creating meta-models. This is the uppermost level in the four-level architecture and called metameta-model. Metameta-models define the ER modelling means used for creating the meta-models (see the Figure 3).

Meta-model objects are instances of metameta-objects.

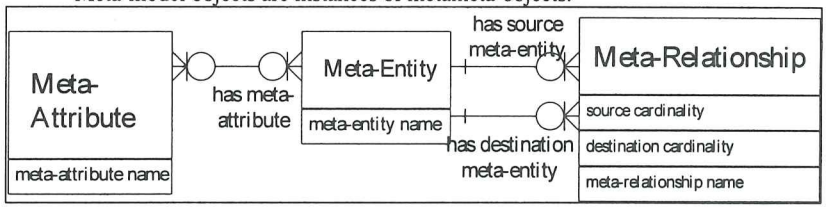

Figure 3. An example of the metameta-model

To summarise up, the following picture can be drawn (Figure 4).

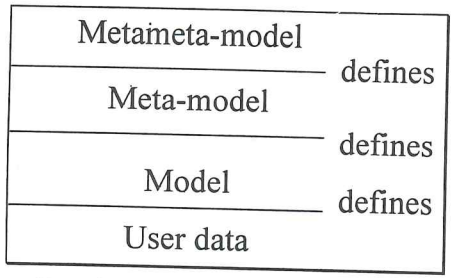

Figure 4. The four-level meta-data architecture

3. Data conversion process

Under data conversion process we assume a process, which takes a source data format file as input and produces a corresponding target data format file as output [2].

In the meta-model data conversion approach a conversion process divides in three subsequent steps:

1. Data import;
2. Logical conversion;
3. Data export.

Figure 5 shows the interaction of the three components of the data conversion system.

Such division of the process allows strictly separate logical data conversion from

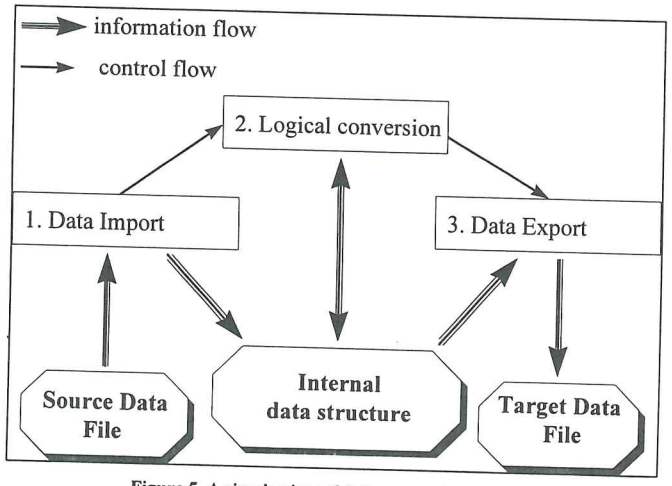

Figure 5. A simple view of data conversion process

technical problems connected with source file format analysis and with target file generation.

Let's have a closer look to each of the components comprising the whole process of the data conversion.

Internal data structure

Internal data structure is used to store internally all the information used during one process of data conversion. The two main concepts used for description of the conversion process data is model and meta-model (see the previous section).

Generally the internal data structure consists of four components:

1. Source data meta-model;
2. Target data meta-model;
3. Source data model;
4. Target data model.

The four components are gradually created during conversion process within the internal data structure.

Initialisation.

Before the conversion process initialisation of internal data structure is performed. Within this step, two meta-models are loaded - one for source data, the other for target data. In the further steps of conversion process those two meta-models will be used for checking of information consistency. That means, that information model, which does not conform to the meta-model, can not be stored in the internal data structure. All the data, which are stored during conversion process in the internal data structure, are instances of source or target meta-model objects.

Description of the role of each conversion component follows.

Data import

Data import is the first step in a data conversion process and is responsible for following functions:

- Parsing of the source data format file [4,5];
- Information store in the internal data structure according to the meta-model of the source data.

Due to the separation of those two tasks, it is possible to use the same syntax of data exchange format for storing different data and vice versa - different data exchange formats may contain the same logical data. This feature is reached because the meta-model defines the logical structure of data, syntax of data exchange format defines only appearance of the data.

For each instance during the data import, the consistency with the source meta-model is checked. If an inconsistency is found, the instance is rejected and an error message appears.

For development of the importer, the development framework offers an application programming interface (API) [3].

After data import, internal data structure contains both meta-models (as it was before import) and model of the source data.

Logical conversion

Logical conversion is the most important step in the whole process of conversion. At the moment when control is passed to the logical conversion module, the internal data structure contains both meta-models and model of source data.

From the operational point of view the logical conversion module performs the same functions with the target meta-model as the importer does with the source meta-model. Logical conversion creates instances in the target model and the same checks are applied also here as in the data import process.

Logical conversion module does not need to know anything about format in which source data were stored. Logical conversion uses the information about source data stored in the source model of internal data structure to produce object instances in target model.

Conversion development framework offers a special-purpose language for logical conversion development - conversion description language (CDL) [3]. This language allows to define what object instances in the target model are to be created using the instances of source model. Description is developed using meta-model concepts of source and target meta-models.

After the logical conversion, internal data structure contains the meta-models and the source and target models. No more modifications in the internal data structure are performed.

Data export

Data export is responsible for the export of target model data produced during the logical conversion. Data exporter is aware of the syntax of target data format and generates an appropriate export file.

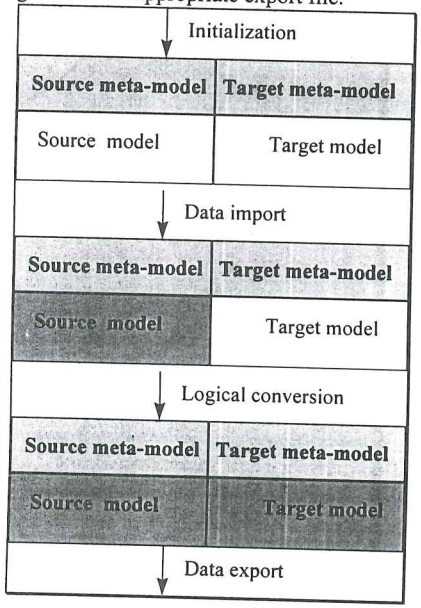

Figure 6. Evolution of the internal data structure during the conversion process

Data exporter has to loop over the target model via instances created during the process of logical conversion. Conversion development framework offers an API for development of exporters [3].

The evolution of the internal data structure during conversion process is shown in the Figure 6. The grey boxes correspond to the filled components of the internal data structure. Initially, all the components are empty.

4. Conversion development framework

This section shortly describes a development framework that is necessary for meta-model based data converters' development.

4.1 Universal metameta-model

Information stored in the converters internal data structure contains models and meta-models. It is sensibly to create internal data structure that is based on meta-model. Such a meta-model will be referred to as *universal metameta-model* (UMMM) further on. The structure of the UMMM has to be available for conversion developer and is especially useful for development of logical conversion. This will make it possible for developer to navigate via the internal data structure using similar notations as in meta-models of source and target data.

What are the basic objects in UMMM ?

Internal data structure contains two models and two meta-models. UMMM has to reflect relationships between source and target meta-models and corresponding models.

Thus, UMMM mainly consists of two types of objects - meta-objects (for source and target meta-models) and their instances. The two types of objects are linked together via relationship <instance, type> (see the Figure 7).

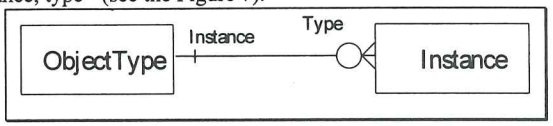

Figure 7. UMMM upper level objects

Each instance (model object) has exactly one type (meta-object). Each type may have zero or more instances.

For further explanation it will be more convenient to separate the UMMM in two parts - meta-model part (let us call it "left" side) and model part ("right" side). These parts will have a lot of common and some different features.

Let us have a closer look to the left side of UMMM (meta-model part). This part defines means for description of meta-models. Thus, the left side of UMMM corresponds to the uppermost level of the four-level architecture (metameta-model). This part should be worked out considering various types of meta-models that will be processed. It is recommended to use the most general type of metameta-model in order to avoid the problems with more complicated meta-models. Examples of some more complicated features of meta-models are as follows - multiple inheritance, n-ary relationships, relationships between relationships, subtyping of relationships. Although all these features can be modelled via simple binary relationships, it is always sensible to keep the real appearance of meta-model.

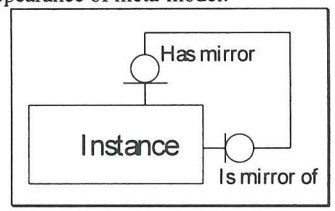

Figure 8. Mirror relationship between instances

The right side of UMMM offers structure for store of source and target models that are created during data import and logical conversion respectively. Basically, the right side of UMMM reflects the same structure as it is in the left side - but instead of object types there are object instances. The main difference is that attribute values are present in model part. If the meta-model tells "object *X* possesses attribute *A*", model contains information "object instance *O* of type *X* has an attribute instance *A* with value *V*".

The other additional feature that is present in model part (right side) of UMMM is relationship of "mirrors"(see the Figure 8). This relationship can be created during logical conversion and links together instances from source and target models. The semantics of the relationship is "the object instance *O* in target model is obtained from the object *O'* in the source model". Such relationship allows to navigate from source model to target model during logical conversion.

Meta-models (including the UMMM) are good for logical description of data. But very often it is necessary to add some information to meta-model which helps for implementation purposes like performance and economy of memory. For such purposes

an implementation meta-model (RUMMM) is used. The RUMMM is not accessible for conversion developer - it is used to implement APIs and CDL.

An example of UMMM can be found in [2].

4.2 Development of meta-models

Before the development of the conversion modules, meta-models for both source and target data must be created. Due to fact that meta-models are a kind of entity-relationship models, a CASE tool should be preferred for development of meta-models. What kind of CASE tool should it be ?

The tool should support all the features that are allowed for the meta-models. In other words, meta-model for of CASE tool must be the same as the left side of UMMM (see the previous section).

However, any CASE tool which supports simple ER-modelling can be used for the development of meta-models. More complex features of meta-models can be modelled via simple binary relationships. If no CASE tool is available, a data format can be defined and the meta-model described using data format. In this case readability of meta-models will be extremely low.

Finally, in the data converter engine, a component, which can load the meta-models from CASE tools or data formats before the start of the conversion, must be present.

4.3 Development of data conversion modules

In this section let us discuss the converter's development environment support for the development of conversion modules.

As it was stated earlier, conversion development framework offers APIs and a special purpose language (conversion description language - CDL) for development of the importers, exporters and logical conversion modules.

Implementation of module (importer or exporter) by an API means implementation of the module using the standard programming language which uses the API's function calls.

It is sensible to use syntax analysis tools [4] for implementation of the data importer modules.

Logical conversion modules are implemented in the CDL language, afterwards they are compiled to the standard C programming language and the executable is obtained using standard compiler. Framework offers the compiler form the CDL to the C.

4.4 Conversion description language

The main task of logical conversion module is to create instances in target model using the information stored in source model. This task very much defines structure of CDL. Description of conversion is based on source and target meta-model objects. Important feature of CDL is backtracking mechanism. This supports automatical looping via all instances of an object type. Developer can select instances in source model using various logical conditions and create corresponding instances in target model. It is also

possible to mix up the elements of CDL with standard programming language elements (for instance, C).

Description of the conversion in the CDL language consists of *rules*. There are two kinds of rules. The first kind of rules is applied automatically to all the data of the source model. The other kind of rules is not performed automatically. They can be invoked from other rules and are executed only when they are called.

CDL rules define what objects have to be created in target model.

In the boundaries of a rule, rule variables can be defined. There are also navigation means present to walk through the source and target meta-models. Navigation can be implemented both using the meta-models' relationships or relationships of UMMM. Let us have a look at a small example of the CDL:

```
$e:lbm_Performer:-eif_ORG_UNIT+eif_NAME{$e.@lbm_Name}+eif_TYPE{aUNIT};
```

This rule acts as follows:

From the source model an instance of meta-entity lbm_Performer is selected. The instance is fixed in the rule variable $e. For the instance $e, an instance of eif_ORG_UNIT is created in the target model. Those two instances are automatically linked together with UMMM relationship of mirrors. Afterwards, an attribute instance of type eif_NAME is created for the newly created instance. The attribute instance gets the value $e.@lbm_Name. This is a CDL expression which returns value of the attribute lbm_Name for the instance which is in rule variable $e. In the same way an attribute eif_TYPE is created. The difference is that here attribute value is a constant.

Such steps are executed for all instances of the meta-entity lbm_Performer in the source model. A more complex example follows, which shows the backtracking mechanism:

```
$e:lbm_DataElement.lbm_Parent.$p($p.@lbm_Structure==aR),
      $e:- eif_DD_FIELD +eif_NAME{$e.@lbm_ElementName},
    {@ [$p.Mirror.eif_DEFINED]eif_R_CONTAINS[$e.Mirror] };
```

In the rule variable $e, instances of type lbm_DataElement are fixed, further the navigation via meta-relationship lbm_Parent is performed and the result fixed in another rule variable $p. In general, navigation may return more than one result. This will make the rule to loop for all instances which are connected via relationship lbm_Parent with that instance fixed in the $e.

Further follows logical condition. Means for the logical conditions are the same as in the C language, extended by the CDL language expression. If the condition is false, the rule is not executed further on. Rule returns execution to the previous rule element, which can give some other value. In this case, it is the navigation via relationship lbm_Parent. If the rule element can not give any more values, rule execution returns to the previous rule element.

If the logical condition is true, the rule is performed further on. The instance which is fixed in rule variable $e, is converted to the instance of eif_DD_FIELD, with the attribute eif_NAME in the same way as in the previous example.

Afterwards, an instance of relationship eif_R_CONTAINS is created. End objects for relationship are specified by the CDL expression in the square brackets ("[ ]"). In general, the end objects may be omitted. In this case, the newly created relationship instance X is connected with mirror objects of the end objects of the mirror of X.

In this example, the end objects are explicitly specified.

In detail description of CDL is presented in [3].

4.5 Implementation of the CDL language

Implementation of CDL compilation is quite complex, and that is the reason why it is divided in two stages. As it was stated above, a language (RDL) for internal use is invented. The language deals with implementation of the internal data structure (RUMMM).

General scheme of implementation of CDL is shown in the Figure 9.

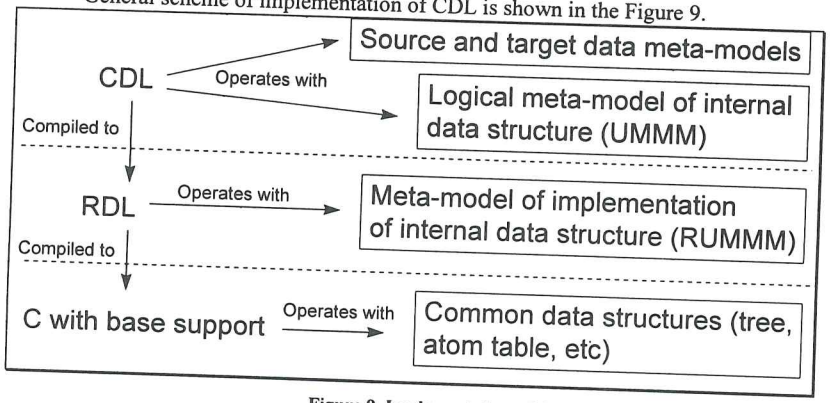

Figure 9. Implementation of CDL

The main idea of such implementation is to separate levels of data implementation by using different means (languages) for each implementation level.

The CDL language is used by conversion developers and that is why the source and target data meta-objects are the most important concepts there. Developer can use the concepts of UMMM for implementation of logical conversion. Actually, use of the source and target meta-model concepts in CDL is not obligatory. They can be described by UMMM notations because UMMM is a meta-model for the source and target meta-models. Nevertheless, for convenience of the developer, they are allowed there.

CDL text is compiled to the language (RDL) which operates with the concepts of implementation meta-model of the internal data structure (RUMMM).

The basic ideology of the RDL language constructions is the same as for the CDL - the main difference is that the RDL operates with RUMMM concepts only.

There is another role for RDL language, as only being an intermediate language for compilation of CDL. Because RDL operates with the internal data structure, it is convenient to use the language for implementation of the importer and exporter APIs. This makes the implementation of APIs more flexible and the code more maintainable because implementation in RDL deals directly with internal data structure. Use of RDL language is not allowed for conversion developer - it is for internal implementation of the conversion development framework.

Further, RDL is compiled to the C language which deals with common data structures like trees, atom table etc. The common data structures are used to implement

RUMMM and base support functions implement elementary operations with RUMMM. The RDL consists only from callable rules and compilation to the C text produces corresponding C functions. The whole implementation schema is shown in the Figure 10.

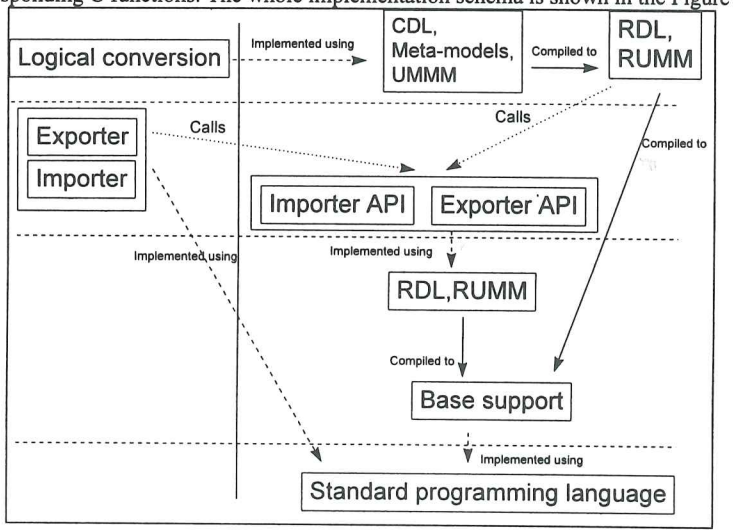

Figure 10. Implementation levels of data conversion modules

5. Application areas

Although the method was initially developed for conversion of CASE tools' data, the method can be used in other areas, too. In fact, method is applicable for any data which has fixed format. Let us have a look at the most common areas where data conversions are necessary and where can possibly arise problems (if any) with the meta-model approach in each of these areas.

5.1 CASE data

This is the "native" application area of the method. CASE data conversion characterises with not too large amount of information. Information, which has to be converted in one piece usually, does not contain more than 10000 objects. This is because models created by CASE tools very seldom may be larger and makes it possible to store all the information created during conversion process in the main memory.

The method and development framework was initially created for model-oriented CASE tools, i.e. such CASE tools which explicitly use meta-models for information store (repository) and metameta-model for import/export format.

5.2 Reverse engineering and code generation

For not model-oriented CASE tools the method was extended and that allowed to process not only CASE data but also workflow tools and programming languages and store the information in meta-models. This allows using the method in reverse engineering area.

The main problem here is to produce a meta-model for information import of the source text of programming language. Programming languages contain information which, if looked in detail, is not very convenient for storing in meta-models. There is no principal problem to create meta-model for storing of, say, arithmetic expressions. But number of objects increases very fast if such information is to be loaded in meta-model. Syntax derivation trees are preferred for storage of information like this.

Meta-model for programming language should be like definition of abstract syntax and need not to define programming language in detail, like formal grammar does.

A good example of use of meta-model approach for reverse engineering is class structure diagram generation from the C++ source code.

The method without major problems can be used for opposite direction - code generation from the CASE tool data. C++ generation from the class diagram structure is a good example here.

5.3 Application data migration

This is the most problematic area for application of the conversion method due to large amount of information. Applications can contain millions of object instances (records) and they can not be imported in the internal data structure due to the shortage of resources. For application data the right method is to convert each object and store the result in the target environment. Thus the application data migration becomes more the task to be implemented in the target environment (DBMS) not explicitly using the meta-model approach.

6. References

1. The CDIF Framework for Modelling and Extensibility, EIA/PN-2387, Electronic Industries Association, Washington, DC, 1993.
2. CASE Data Converter for Windows 2.1 - Software AG 1995
3. CACATOO 3.0 Developer guide - Riga Information Technology Institute 1997
4. Levine J.R, Mason T., Brown D., LEX and YACC, O'Reilly & Associates,, 1992.
5. Aho A.V., Ravi Sethi, Ullman J.D., Compilers, Principles, Techniques and Tools , Addison-Wesley, 1988.
6. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modelling and Design, Prentice Hall, Englewood Cliffs, N.J., 1991.

Towards a Metamodel-Based Universal Graphical Editor

Ugis Sarkans, Janis Barzdins, Audris Kalnins, Karlis Podnieks

Institute of Mathematics and Computer Science,

University of Latvia

Raina bulv. 29, LV-1459, Riga, Latvia

e-mail: {usarkans, jbarzdin, audris, podnieks}@cclu.lv

Abstract

In this paper a novel idea of describing and building universal graphical editors is described, based on advanced metamodeling techniques. Efficiency of the proposed approach is demonstrated for applications in the fields of business modeling and CASE, but it could be used for graphical database browsers as well. With our approach the universal editor would, on one hand, be able to offer a rich set of editor services, like the existing GRADE graphical editors, and, on the other hand, be customizable for use with different sets of modeling concepts.

1. Introduction

Two approaches to building graphical editors can be distinguished so far: universal editors permitting complete freedom over what can be drawn and how to interpret what has been drawn, and graphical editors used in, e.g., environments of CASE or visual programming languages that impose graphical syntax over visual patterns, at the same time offering additional services, like name prompting, syntax checking, creating and navigating links between different diagrams and so on.

We have experience in building the second kind of graphical editors, for use in the framework of GRADE modeling language [3]. GRADE is a business modeling and CASE tool that supports a fixed set of diagram types used in modeling, as business process diagrams, orgcharts, UML-conformant class diagrams and some other. Each of the diagram types is supported by the corresponding graphical editor that offers automatic layout capabilities, cut-and-paste facility and other features usual for graphical editors, as well as some functionality pertaining to the semantics of the corresponding diagram type, as syntax checking and mouse-enabled navigation between related diagrams.

2. A Universal Graphical Editor - Motivation

Our discussion will be structured around the notion of metamodels, and in the Figure 1 a fragment of the GRADE language metamodel is presented. The term "metamodel" usually is referred to a diagram(s) that defines the language for specifying models, just like a model defines a language to describe an information domain [10].

Now suppose that for some reason we want to change the underlying metamodel, either to add additional concepts or change the associations between the existing ones, or both. For example, we could wish to enrich the GRADE language by adding concepts from the top management domain, like measure of performance or business perspective, which are missing not only from the metamodel fragment presented here; the current version of GRADE does not support these concepts. In the current situation that would mean complete redesign of not only the GRADE language, but also the corresponding toolset, i.e., graphical editors.

One of the solutions that would not necessitate redesign of graphical editors would be using the current diagrams and graphical elements with different semantics. GRADE users occasionally have done that, and there are both positive and negative experiences. Still, for this approach to work the new concepts and relations among them have to be isomorphic to the existing concepts and relations directly supported by diagrams and graphical elements under question, severely restricting the language extension possibilities. Secondly, such style of using graphical primitives with semantics other than what they usually mean is hard to document, and, if we would like

to export models from the GRADE repository and import them into some other repository, we would have to make special provisions regarding exporting and importing objects with (implicitly) modified semantics, therefore the problem would be just shifted from graphical editor design to repository organization issues.

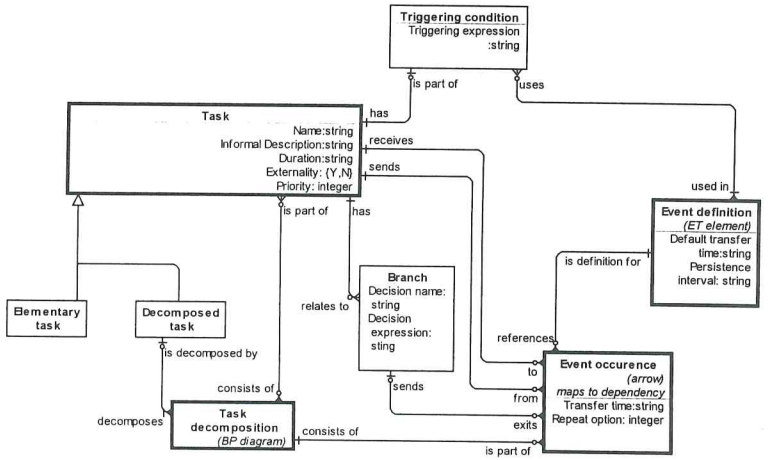

Figure 1. GRADE metamodel fragment.

Another solution would be using the universal diagram type - class diagram, which also can be used for drawing class instances - to model system parts that are not directly supported by specialized, easy to use editors. But this is not a very convenient method for drawing models that are not trivially small. Besides, the repository organization problems would arise here as well - how to integrate parts of the model drawn using the usual diagrams with the new parts modeled in class (and instance) diagrams.

We propose development of a universal graphical editor that can be easily, without programming, customized to suit needs of any metamodel-described graphical language. Our 7-year experience in implementing advanced graphical editors makes us believe that this is a feasible task, although not an easy one. A more correct name for the universal graphical editor would be "universal graphical editor framework", because

it can not be used directly, without customization. Still we will use the shorter version when referring to it.

Several other meta-level graphical editors, e.g., EDGE [7], TGE [5], MetaEdit [9], VSF [2, 8], GEDL [4], Hardy (University of Edinburgh), as well as general models, e.g., Entity-Aggregate-Relationship-Attribute with Graphical Extension (EARA/GE) Data Model [1], have been developed. For some of these tools customization facilities seems to be quite *ad-hoc*. Our metamodel-based approach is much more uniform, being able to encode graphical layout information as well as tool behavior aspects.

3. Pure semantic metamodel and Syntactic-semantic metamodel

The description of our approach will be based on examples that will describe GRADE (both language and tools). One version of GRADE metamodel fragment was already presented above.

We will assume for a moment that we have no tools for GRADE; the only thing we have is a language metamodel, a part of which was presented above.

This metamodel does not provide any information regarding how the GRADE model should be structured, what graphical elements should be used and what services provided. For example, we can see from the model that a Task can receive zero or more Message event occurrences, but there is no information as to how this fact has to be rendered in diagrams. Actually there is no notion of diagram present in this metamodel. We will call such metamodels pure semantic metamodels.

In order to make this metamodel understandable to the (still hypothetical) universal graphical editor, we will extend and transform it using annotations carrying the information about how the objects have to be drawn and how they should behave in response to user events, like mouseclicks. We will use the UML notion of stereotypes for purposes of this annotation. The annotated metamodel will be called syntactic-semantic metamodel.

3.1 Class annotations

First, we will add a stereotype to each class in the pure semantic metamodel. The stereotypes will describe how the corresponding class instances have to be drawn, and they will be taken from a fixed set understandable to the universal graphical editor. A fragment of GRADE syntactic-semantic metamodel that corresponds to the fragment of the pure GRADE semantic metamodel presented above is given in Figure 2.

The universal graphical editor will understand how to display the following graphical primitives:

«Box (Rectangle)»	used for tasks (both elementary and decomposed) in the example metamodel
«Box (Hexagon)»	for branches
«Arrow»	for event occurrences
«Table Row»	for event definitions
«String»	for triggering conditions
«Diagram»	for task decompositions

For complete description of possible stereotypes a metameta-level model is needed where classes correspond to possible graphical (and some other – see below) stereotypes.

3.2 Association annotations

Associations in the original pure semantic metamodel will be converted to classes, and stereotypes will be added to describe how the universal graphical editor has to interpret associations. We will call classes in the syntactic-semantic metamodel obtained from associations in the pure semantic metamodel association classes.

As an example (see Fig. 1 and 2), we can read from the pure semantic metamodel that a *Decomposed task is decomposed by 1 or more Task decompositions*. In the syntactic-semantic metamodel this association is converted into a class *is decomposed by*, and associations connect this class to classes *Decomposed task* and *Task decomposition*. Associations in the syntactic-semantic metamodel have only two names - "out" and "in". The construct involving association class *is decomposed by* has

to be read from the end of "out" association: *Decomposed task / is decomposed by / Task decomposition.*

Stereotype «Refinement» is attached to the association class *is decomposed by*. The semantics of this stereotype from the point of view of the universal graphical editor is such that if the user clicks on a Box (Rectangle) corresponding to some *Decomposed task* instance, a new window containing a Diagram for the corresponding *Task decomposition* has to be opened (if there are several corresponding *Task decompositions*, the user has to be asked first which one to open).

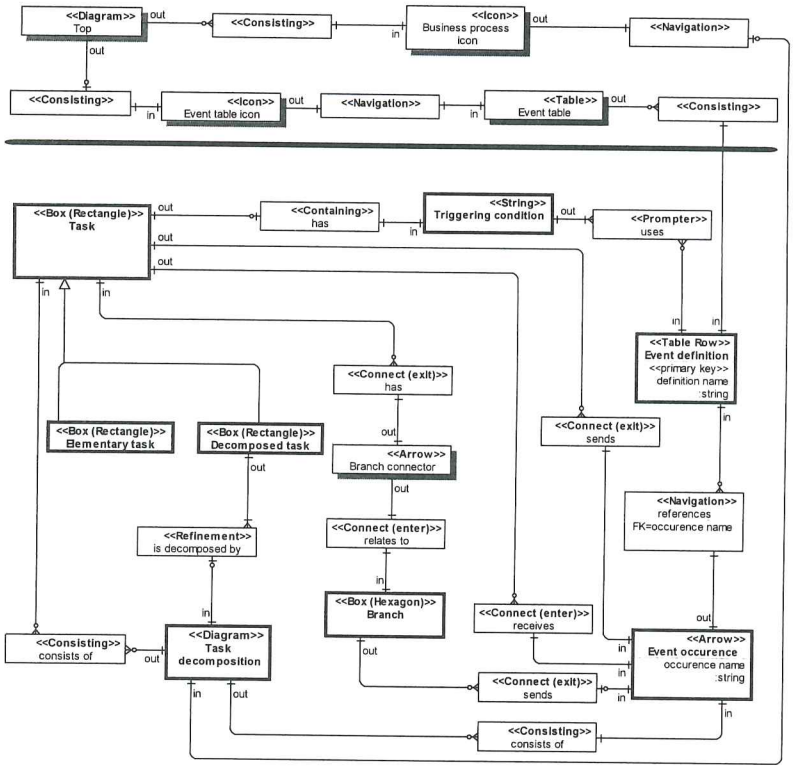

Figure 2. GRADE syntactic-semantic metamodel fragment.

Short description of other association class stereotypes present in the sample syntactic-semantic metamodel:

«Navigation»	similar semantics to «Refinement», only the target object does not have to be a complete diagram, it can be a part of some diagram, like a table row in the example
«Consisting»	connects a diagram to graphical elements this diagram has to include
«Containing»	connects a graphical element to some other graphical element that has to be displayed inside the first one
«Connect (exit)» and «Connect (enter)»	connects an Arrow to Boxes
«Prompter»	used for describing name prompting service, as in the example where the user can be prompted which Event definitions can be used in a Triggering condition String

We convert associations in the pure semantic metamodel to classed in the syntactic-semantic metamodel because of several reasons:

- in the original, pure semantic metamodel association names can be arbitrary, and they are retained in the corresponding syntactic-semantic metamodel in the form of names of association classes; however, only one association name is retained if an association is converted into one association class;
- addition of attributes is sometimes necessary for precise description of (original) associations.

There are two kinds of association from the perspective of graphical editing – those describing links that are directly established in the editing process and those describing links that are implied / computed. In our example, the association *Decomposed task - Task decomposition* is of the first kind, while the association *Event occurrence / references / Event definition* is of the second kind - an instance of *Event occurrence* can be defined by directly typing in its name, and the reference to the corresponding instance of *Event definition* is computed by establishing name equivalence. The example syntactic-semantic metamodel illustrates use of association class attributes in *references* class. The attribute *FK* (foreign key) defines which attribute of the *Event occurrence* class has to be compared to the primary key attribute of the *Event definition* class when establishing links between instances of *Event*

occurrence and *Event definition*. Existence of a primary key attribute in the target class is an additional requirement if we want to use this simple «Navigation» facility, and in the example metamodel the primary key attribute is marked with attribute stereotype «primary key».

We could wish to provide along with the «Navigation» service from *Event occurrences* to *Event definitions* also a «Prompter» service in the opposite direction. Then we would draw an association class with stereotype «Prompter» and connect it to *Event occurrence* and *Event definition* classes in parallel with the «Navigation» - *references* class. Therefore an association in the pure semantic metamodel is not always converted to exactly one association class.

Another example of translation from associations to association classes that would be hard to formalize is demonstrated by the association between *Task* and *Branch* in Figure 1 and the corresponding chain of 3 classes in Figure 2. The reason for this is that the association in the pure syntactic metamodel corresponds to graphical objects - links between *Task* Boxes and *Branch* Boxes, rather than encoding some kind of logical relationship. In this case a new object (i.e., non-association) class is introduced («Arrow» - *Branch connector*), and both names of the original association between *Tasks* and *Branches* (*has* and *relates to*) can be retained in the corresponding two association classes.

3.3 Auxiliary parts of syntactic-semantic metamodel

The universal graphical editor needs to know what to do when the user wants to start navigating or editing some model. For this purpose some more auxiliary classes has to be added to the syntactic-semantic metamodel (shown above the vertical bar in Figure 2). One of these classes should have predefined name *Top* and stereotype «Diagram»; this will be the entry point into the model, the diagram *Top* being the initial diagram appearing when the user opens the corresponding model. Some more new class stereotypes appear in the auxiliary part of the metamodel in Figure 2:

«Icon»	used to represent the *Event table* and business process diagrams (*Task decompositions*) in the *Top* diagram
«Table»	used for grouping *Events* in the *Event table*

Some of the classes in the auxiliary part of the syntactic-semantic metamodel are shown without class names. This is done on purpose to emphasize that the most important information about these classes is encoded in their stereotypes; to make such a metamodel conformant to UML syntax we can introduce some placeholder names for these classes.

In GRADE the top diagram contains a model tree. Such constructs are also possible to describe using our formalism of syntactic-semantic metamodel, but for illustration purposes we assume that the icons in the Top diagram are unrelated.

4. Application of the syntactic-semantic metamodel

The process of translating a pure semantic metamodel into a syntactic-semantic metamodel is an intellectual task - it cannot be automated, unless we want to impose some (heavy) restrictions on how the pure syntactic metamodel can be built. One pure semantic metamodel can be translated into several syntactic-semantic metamodels by adding different auxiliary classes and associations, representing original classes and associations by different graphical elements and editor services, grouping class instances into diagrams in different ways. Once the universal graphical editor is built, the different syntactic-semantic metamodels can be compared to determine what is the best way of providing tool support for the given set of semantic primitives.

After the translation between the two types of metamodels has been established, automatic two-way translation between a pure semantic model and a syntactic-semantic model is possible. The universal graphical editor, using the syntactic-semantic metamodel, is able to present the (instance) syntactic-semantic model to the user, interpret user actions and change the model. Automatic translation between two kinds of metamodels ensures that the syntactic-semantic model can be converted to the pure semantic model, e.g., for exchanging the information with other CASE tools capable of supporting the particular pure semantic metamodel (see Figure 3).

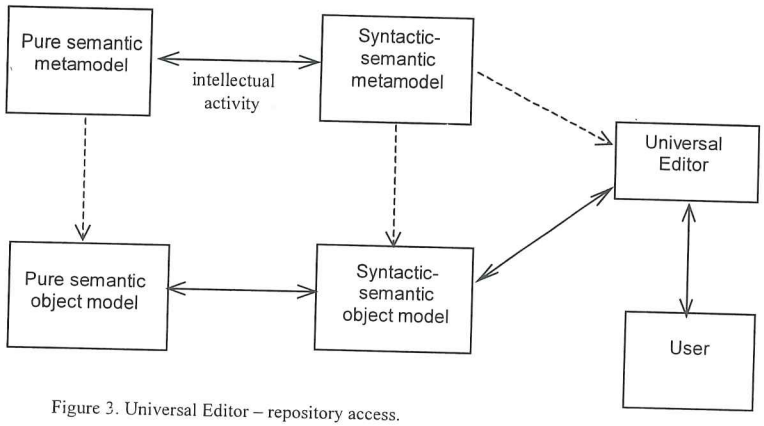

Figure 3. Universal Editor – repository access.

5. Conclusion

Implementation of the universal graphical editor is feasible; most of the necessary components have already been developed for GRADE. The crucial difference between GRADE and the universal graphical editor is that while in GRADE these components are linked in a fixed structure supporting the GRADE language, for the universal graphical editor a more flexible way of structuring the components according to the metametamodel of stereotypes is necessary.

The universal editor customized using, for example, the syntactic-semantic GRADE metamodel could be not as good as specialized GRADE editors because of some performance overhead and difficulties of describing some very fine features of native GRADE editors using this formalism. Therefore it is not possible to state that such universal graphical editor can replace GRADE editors or any other graphical editors. Still such universal editor can be used for many purposes, like quick prototyping of several possible configurations of editor sets (see above about multiple possible translations between models) or filling the repository built according to some metamodel that is not yet supported by any graphical tools.

References

[1] D.Gadwal, P.S.Findeisen, P.G.Sorenson, J.P.Tremblay, B.L.Millar. "Generating Customizable Software Specification Environments Using Metaview", research paper, Department of Computational Science, University of Saskatchewan.

[2] M.Heym and H.Osterle. Computer-aided methodology engineering. *Information and Software Technology, 35 (6/7), 1993,* pp. 345-354.

[3] A.Kalnins, J.Barzdins et al. Business Modeling Language GRAPES-BM and Related CASE Tools. In *Proceedings of the Second International Baltic Workshop on Databases and Information Systems, Tallinn, 1996, v.2,* pp. 3-16.

[4] A.S.Karrer. Generating Graph Editors. Ph.D. thesis, Computer Science Department, University of Southern California, May 1993.

[5] A.S.Karrer and W.Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *ACM SIGGRAPH Symposium on User-Interface Software and Technology, October 1990,* pp. 84-92.

[6] R.K.Keller, M.Cameron, R.N.Taylor, and D.B.Troup. Chiron-1: a user interface development system tailored to software environments. Technical Report UCI-90-06, Department of Information and Computer Science, University of California, Irvine, June 1990.

[7] F.Newbury Paulisch and W.Tichy. EDGE: an extensible graph editor. *Software Practice and Experience, 20 (S1), June 1991.*

[8] J.N.Popcock. VSF and its relationship to open systems and standard repositories. In *Software Development Environments and CASE Environments, Springer Verlag, LNCS Vol. 509, 1991,* pp. 53-68.

[9] K.Smolander, P.Marttiin, K.Lyytinen and V-P.Tahvanainen. MetaEdit – a flexible graphical environment for methodology modelling. In *Advanced Information Systems Engineering, Springer Verlag, LNCS Vol. 498, 1991.*

[10] UML Semantics. Version 1.1, 1 September 1997. http://www.rational.com/uml

Application generation for the simple database browser based on the ER diagram

Guntis Arnicans

University of Latvia
Faculty of Physics and Mathematics
Rainis Blvd. 19, Riga LV-1459, Latvia
garnican@lanet.lv

Abstract

This paper describes a development technique for the rough browser of a database. The offered data browser or data management system can be generated automatically from the a physical data model represented by an ER diagram. The ER diagram used to generate a target application source text is described by common simple concepts and by some additional attributes with default changeable values. All the ER diagram elements are mapped to standard screen object groups and are the main components in the target system screens. Various screen templates for generated applications are defined depending on the entities, the relationships between them and an acceptable user interface. The generated application can be used for database browsing, data manipulating, system prototyping, fast developing of simple information systems and data analyzing.

1. Introduction

There are many strategies for information system development and project management in nowadays. The development of very advanced CASE tools lets us use the Rapid Application Development (RAD) methodology. This approach includes several steps - business modeling, data modeling, process modeling, application generation, testing and turnover [1]. In this paper the simple technique is described that allows to develop specific information system - database browser and data manager. The main attention is turned to the generation of application.

Many powerful tools already exist to assist in system development with RAD technology, for instance, Oracle Designer/2000 [2], [3]. But practice shows that these tools sometimes are not useful. The reasons are that they are expensive and require high educated and trained specialists to work with them. And we have to work hard for some

time. The quality of the information system mostly depends on the data model. Especially this data model (object model) is critical when we use Object Modeling Technique (OMT) [4]. Let us assume that we already have designed the physical data model for our database. Like a conceptual data model the physical data model can be described by Entity-Relationship Diagram (ER diagram). This is popular instrument to describe data model or database and these diagrams are known for most programmers.

Our goal is offer to the user a technique that allows to create a database browser from the physical data model described by the ER diagram. What does the developer have to do? He has to create a simple ER diagram for the existing or the planned database. We do not care in whether he makes a serious analysis and design, whether he creates the ER diagram "on the fly". He obtains quickly generated database browser, a simple information analysis and filtering tool, a data entering and editing tool, a prototype for the most serious business application, simple database testing tool. Thus, while the real system is developed, a robust information system is obtained.

2. ER diagram - the source for generation

2.1 The elements of the ER model

The ER diagram is a source for application generation. We consider only the ER diagrams that represent physical data models. The main objects we manipulate are the ER diagram descriptor (describing common features of database), the entities (representing tables in our database), the relationships (representing the relations between the entities), the fields (representing the data fields in the record of the physical table).

Developers use various variants of the ER diagrams. Let us take a diagram that is not too simple and not very complex. The ER diagram can be described by the object model shown in the Figure 1. We choose the following elements in the ER diagram:

- **diagram descriptor** - *DiagramName*;
- **entity** - *EntityName*, [*EntityType*], *PrimaryKey*, *UniqueKey*, *Index*;
- **field** - *FieldName*, [*FieldType*], *DataType*, [*Visibility*], [*ShortView*], [*LongView*];
- **relationship** - *EndEntity_1*, *EndEntity_2*, *Cardinality_1*, *Cardinality_2*, *Role_1*, *Role_2*, *ForeignKey_1*, *ForeignKey_2*.

The attributes in brackets are introduced for generation better applications. For simplicity we assume that primary key and foreign key are represented only by one field. In our simplified model we assume that an index is created from a field without using any function. It is not so hard to expand the model to use a combination of fields as the primary key (as the foreign key respectively) and a function of fields as the index.

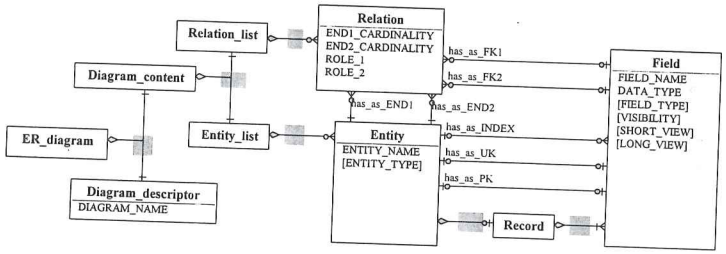

Figure 1 The conceptual object model for the ER diagram

2.2 Additional elements of the ER diagram

Let us introduce several new attributes for the ER diagram. These attributes provide the additional information for the application generation program to generate a more convenient application.

Entity type is a special attribute of an entity that allows us to generate application screens with a specific information layout and data manipulation means. This attribute is stated automatically and depends on the relationships between the entities. The user can correct it while automatic type fixing.

Field type is an automatically calculated attribute of the field. If the field is defined as the primary key of *Entity* via relationship *has as PK* (Figure 1) then the field has type ***PK***. Similarly we define type ***UK*** (via relationship *has as UK*) and type ***FK*** (via relationship *has as FK1* or *has as FK2*). Otherwise the field type is ***Attribute***.

Visibility is a feature of a field. It states whether the information associated with the field is or is not displayed to the user. The default value of *visibility* is TRUE for fields with types *UK*, *FK* and *Attribute* but FALSE for type *PK*.

ShortView and **LongView** are field attributes that define how the entity record can be best displayed on the screen.

2.3 Entity types

Entity type is an important concept in our application generation ideology. Let us define the following *entity types*.

- ***Domain*** - list of standard data elements, allowed values for an attribute or an object property.
- ***SimpleEntity*** - simple object that is determined by a set of attributes or standard data elements.
- ***ComplexEntity*** - complex object is similar to *SimpleEntity* but it includes other simple or complex objects.
- ***Link*** - logical relation between at least two simple or complex objects.

2.4 Algorithm for entity type determination

1. Scan through all the entities and fix those that have no field with type *FK* (foreign key) and all the incoming ends of whose relationships are either of cardinality 1 or 0..1. We have to assign the type *Domain* or *SimpleEntity* to the fixed entities. The type *Domain* is assigned by default.
2. Scan through all entities without a fixed type and fix any one that has fields with type *FK* referenced only to entities with type *Domain* and all the incoming ends of whose remaining relationships are either of cardinality 1 or 0..1. The type *SimpleEntity* is assigned to the fixed entities.
3. Scan through all entities without a fixed type and fix each one which at that moment is referenced by a foreign key from any entity with type *SimpleEntity*, *ComplexEntity* or undefined type. The entity can also reference to itself. The type *ComplexEntity* is assigned to the fixed entities.
4. Scan through all entities without a fixed type and fix any one that has at least two fields with type *FK* that reference to entities with type *SimpleEntity* or *ComplexEntity*. We have to assign the type *Link* or *ComplexEntity* to the fixed entities. The type *Link* is assigned by default.
5. The type *ComplexEntity* is assigned to the any remaining entity without fixed type.

The user decides on the entity type (1. and 4. step) according to the semantics of the entity and on how he wants to see the information on the screen.

2.5 Textual visualization of entity record

We need to define several textual visualizations of an entity record in our system. A textual visualization of an entity record is mapping the field values to text. These texts (or entity views) are used to display an entity on the screen. Let us define three functions: *shortView()*, *longView()*, *allFields()*. The *shortView()* displays some of the record fields in an ordered sequence. The order is defined by assigned an order number to attribute *ShortView*. If the field is not included in short view the 0 is assigned to *ShortView*. The similar approach is used for *longView()*. The *allFields()* displays all fields.

2.6 ER diagram for example

The ER diagram for example is given in Figure 2. The attributes of each field are given in the following sequence - *field name*, *data type*, *field type*, *visibility* (T for TRUE, F for FALSE), *ShortView*, *LongView*.

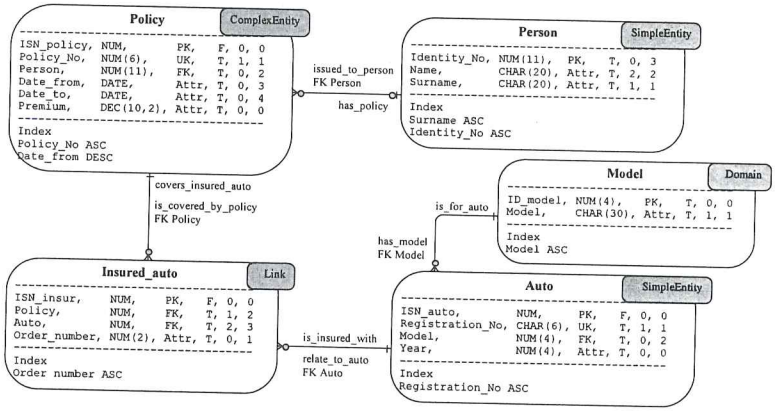

Figure 2 ER diagram for example

3. Application generation

The general idea of generation is to create a system with a predefined user interface and functionality. The features of the system depend on the generator. We can generate the whole application for the ER diagram or only some components for this application.

3.1 Screens and menu system

The quality and usefulness of the generated system depends mainly on the generated screen system. Let us define several standard screens that allow us to handle data in the database. The basic generation principle in our approach is to generate the specific data editor for one or several tables connected by relationships. We generate a set of related screens and this enables direct transition from one screen to another.

The primary objects in our system are entities and relationships between them. We define some screen types with different user interface and different functionality for any entity or relationship. For instance, we can display on the screen entity, links to other entities (relationships in the ER model), information about related entities or display related entities by some relationship. The screens of all types are generated for each entity (accordingly to entity type) and for each relationship if the generation options do not define another behavior.

The menu provides access to any generated screen and standard operation defined for any application. The screens are organized in a hierarchy for easy orientation.

3.2 Screen components

Every screen logically consists of two large sets with different screen objects.

- **Information group** - screen objects that display information stored in the database and objects that are generated from the ER model. This group mainly consists from table fields and relations between entities. The basic *information subgroups* are: **Field group** (screen objects that display visible record fields), **Entity presentation group** (screen objects that display the record of the entity or the list of records), **Relationship presentation group** (screen objects that display relationship between entities), **Order button group** (radio button group that determines the order in what records are ordered).
- **Management group** - screen objects that provide additional management over the data stored in the database. Their generation depends on the screen type. The following *management subgroups* are defined: **Edit button group** (a group of buttons for entity record editing - *New*, *Edit*, *Save*, *Delete*, *Cancel* buttons), **Locate button group** (a group of buttons for locating the desirable record - *First*, *Next*, *Previous*, *Last*, *Find* buttons), **Print button** (a button for printing the current record

or a record list), **OK button** (a button for leaving the window), **Control button group** (specific buttons included in the screens of specific type).

4. Mapping ER model objects to application objects

4.1 Mapping sequence

A rough algorithm for application generation is the following.

- Map the ER diagram name to application name.
- Generate the screens for each entity (all screen types allowed for given entity type):

1. Generate screen name from entity name.
2. Generate each information subgroup needed for the given screen type. The layout of this group (horizontal, vertical, tabular or other) is not the subject of this paper.
   a) Generate all field groups for given entity.
   b) Generate information about all related entities via foreign keys.
   c) Generate information about all related entities via relationship without a foreign key at the end of given entity.
   d) Generate order button group for given entity.
3. Generate all management subgroups necessary for the given screen type.

- Generate screens of all allowed types for each relationship.

1. Generate screen name from related entity names and role names.
2. Generate information group as for the entity screens.
3. Generate management group as for the entity screens.

- Generate menu system organized by screen types. The deepest menu items are generated from the entity name (for entity based screen) or from two entity names and role names (for relationship based screen). The menu item will open the window for the specified screen type and entity (or relationship) as the central object.
- Generate additional menu items for standard operations.

4.2 Field mapping

A field with type *Attribute* maps to screen object group ***AttributeInfo*** (Figure 3).

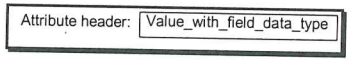

Figure 3 Screen object group AttributeInfo

Attribute header is TextBox object with value generated from *FieldName* and EditBox object contains the value with type defined by field *DataType*. The EditBox length depends on the data type but is limited by some reasonable maximal length. If necessary scrolling through field is provided.

A field with type *PK* (primary key) maps to screen object group ***PkInfo*** (Figure 4). It is similar to *AttributeInfo* but it also has the button ***Gen***. The user can manually enter a value for the record primary key or generate a value automatically by pressing the button *Gen*. Key generation depends on the selected default rule. When the user leaves EditBox the system checks whether the value is unique.

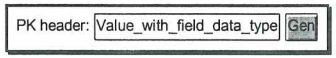

Figure 4 Screen object group PkInfo

A field with type *UK* (unique key) maps to screen object group ***UkInfo*** (Figure 5). This group is similar to the screen object group *PkInfo*.

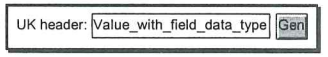

Figure 5 Screen object group UkInfo

A field with type *FK* (foreign key) maps to screen object group ***FkInfo*** (Figure 6). The content of this group depends on the screen type, relationship type and connected entity type. *Fk header* is TextBox object with value generated from *FieldName*. CheckBox is an optional element and is generated when corresponding relationship at the opposite end has cardinality 0..1, otherwise (cardinality is 1) CheckBox is not generated. We can assign an empty value to the foreign key field by turning off CheckBox. *EntityInfo* is another screen object group (see Entity presentation). The button ***Go*** is optional and its generation depends on the screen type.

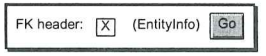

Figure 6 Screen object group FkInfo

4.3 Entity presentation

An instance of entity is represented by screen object group ***EntityInfo***. Let us define several subgroups for *EntityInfo*.

Entity with type *Domain* is represented by ***DomainInfo*** (Figure 7). ComboBox provides the selection of domain value and shows the current value. This screen object is obligatory part of the group. EditBox *Domain_key* is optional. It is generated by the following rules. At first, if entity has the visible unique key then *Domain_key* gets the data type from this field.

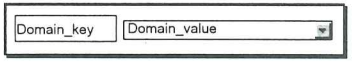

Figure 7 Screen object group DomainInfo

Otherwise, if entity has the visible primary key then it gets the data type from this field. If entity has no visible unique or primary key then EditBox is not generated. Both objects always reference to the same table record. EditBox can be used for selecting the domain value by entering the key in the EditBox. *Domain_value* is the entity representing text depending on the default text function.

Entity with type *SimpleEntity* or *ComplexEntity* is presented by ***EntityTextInfo*** (Figure 8). TextBox contains the entity representing text depending on the default text generation function.

Figure 8 Screen object EntityTextInfo

Several similar entities with type *SimpleEntity* or *ComplexEntity* are presented by ***EntityListInfo*** (Figure 9). ListBox contains entities representing texts depending on the default text generation function. *EntityListInfo* represents also entities with the type *Link* but in the text generation function can exclude one field with type *FK*.

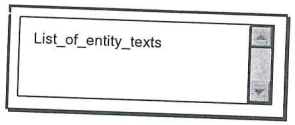

Figure 9 Screen object EntityListInfo

4.4 Relationship representation

One direction of relationship is represented by ***RelationshipEndInfo*** (Figure 10). Let us suppose that we represent relationship from *Entity_1* to *Entity_2* with role *Role_1*. TextBox *Relation_role* contains text 'Role_1' and TextBox *Relation_end* contains text 'Entity_2'. Button ***Go*** provides going to the screen that represents entity *Entity_2*.

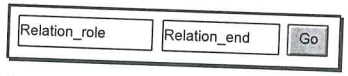

Figure 10 Screen object RelationshipEndInfo

4.5 Mapping of the indexes

If the entity has at least one index then all indexes are mapped to Order button group represented by ***OrderButtonGroup*** (Figure 11). The first button *None* allows to remove any previously used record sequence. Every next button corresponds to some index.

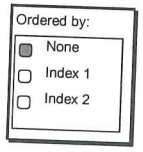

Figure 11 OrderButtonGroup

4.6 Traversing through screens

Traversing through screens is performed by button ***Go***. This button is usually attached to the screen object group representing the entity. The button brings us to another screen that belongs to this entity. The button can work in two modes - with filter or without one. If the filter option is chosen then in the newly opened screen we can access only those records that are logically tied with the record or records in the previous screen. For instance, if we fix any car model in the table Model then in the table Auto only cars with this model are accessible.

5. Screen types

The design of screen types depends on the user's needs. Let us define screen templates that can be regarded as basic screen types. The screen examples correspond to Figure 2.

- **Simple entity view**

This screen type can be used for the representation of any entity (Figure 12). The information group contains all visible field groups and *OrderButtonGroup* if any index is defined. The management group contains Edit button group, Locate button group, Print button, OK button.

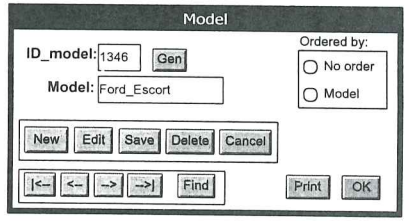

Figure 12 Screen for entity Model

- **Entity view extension with links**

This screen extension can be added to screens of entities with type *SimpleEntity* or *ComplexEntity*. All entities that have type *Link* and are directly connected via relationship with the given entity are shown on the screen. The presentation is performed by screen object groups *RelationshipEndInfo* and *EntityListInfo*. Figure 13 contains a screen fragment for the entity Policy with insured cars (*LongView* option is used) for the current policy.

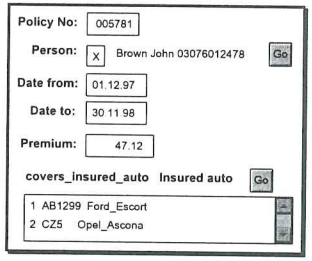

Figure 13 Fragment of screen for entity Policy

- **Entity view extension with relations**

This screen extension can be added to the screens with types *Domain*, *SimpleEntity* or *ComplexEntity*. We represent all relationships that have foreign key at the opposite end (it means that the other entity references to the given entity via foreign key) and the corresponding entities. For the presentation screen object groups *RelationshipEndInfo*, *EntityTextInfo* or *EntityListInfo* are used. All the policies (*LongView* option is used) are displayed for the current person in Figure 14.

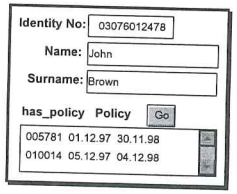

Figure 14 Fragment of screen for entity Person

- **Simple link view**

This screen is the special view to the entities with the type *Link* that links together exactly two entities with type *SimpleEntity* or *ComplexEntity* (Figure 15). The *ShortView* option is used to represent linked entities in *EntityListInfo*. A special *Control button group* determines the main ListBox. In this case for a fixed policy 005781 all insured cars are displayed in the other ListBox with label Auto.

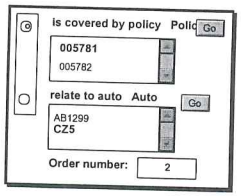

Figure 15 Fragment of screen for entity Insured auto

- **Embedded entities view**

This screen type is useful only for the entity with type *ComplexEntity*. Let us take *Simple entity view* as a base for such an entity. Instead of each field group *FkInfo* we incorporate all visible fields from the related entity. We can imagine each embedded entity as a subwindow where it is displayed with *Embedded entities view* for complex entity or *Simple entity view* for the simple entity. E.g., in Figure 13 the objects group with header *Person* is replaced by three screen object groups - *Identity_No*, *Name*, *Surname* from entity *Person*. During the generation process we must beware of cyclic embedding and stop embedding when we discover the cycle.

- **Relationship view**

This screen provides a special view to relationship and entities connected by it. Related entity is represented by *RelationshipEndInfo* and *EntityListInfo*. The main entity is selected by the radio button. Figure 16 shows the fragment for relationship [Model] is_for_auto / has_model [Auto] with *ShortView* option.

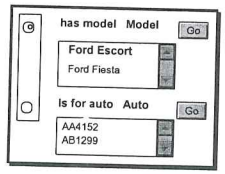

Figure 16 Fragment of screen for relationship between entities Model and Auto

6. Conclusion and future directions

This approach is based on the common ER diagram elements mapping to some screen object constructs. It is not hard to create templates for generation - the standard code for the whole screen and the standard SQL based code fragments for each generated screen object group. Generation basically is code compiling from prepared code templates. This technique partly is applied in practice - the real business applications are developed but screen code is written by hand.

This approach has several future directions that seem very interesting. The screens can be generated dynamically while application is running. E.g., the appropriate HTML page can be generated and displayed. This improvement enables the information view to be changed dynamically.

An ER diagram can be described by context-free grammar (E.g., in BNF notation). The generator is an interpreter that reads the ER diagram as a program and creates a source code for the screens [6]. Other graphical tools, e.g., GRADE [5] can be used to prepare the ER diagram as input statements according to this grammar for the generator.

7. References

[1] Tucker, A.: The Computer Science and Engineering Handbook, CRC PRESS, 1997.

[2] Billings, C., Billings, M., Tower, J.: Rapid Application Development with Oracle Designer/2000, Addison-Wesley, 1997.

[3] Anderson, W., Wendelken, D.: The Oracle Designer/2000 Handbook, Addison-Wesley, 1997.

[4] Rumbaugh, J.: Object-Oriented modeling and Design, Prentice-Hall, 1991.

[5] Barzdins, J., Kalnins, A., Podnieks, K. et al.: GRADE Windows: an Integrated CASE Tool for Information System Development, Proceedings of SEKE'94, pp.54-61, 1994.

[6] Arnicane, V., Arnicans, G., Bicevskis, J.: Multilanguage Interpreter, Proceedings of the Second International Baltic Workshop, pp.173-174, 1996.

Geographic Information Retrieval with Loosely Integrated Information Systems

Gergely Lukacs
Department of Measurement and Information Systems
Technical University of Budapest
H-1521 Budapest, Hungary; lukacs@mmt.bme.hu

Abstract

Geographic Information Retrieval (GIR) is concerned with searching and retrieving geographically relevant data objects, which are typically unstructured or semi-structured textual documents. GIR is required by application areas such as leisure-time activities, commerce or public administration.

Present approaches, applied in traditional and digital libraries, have the following major limitations: (1) They focus on indexing, and neglect query evaluation. (2) They make no or very limited use of formal spatial data models, which is a precondition for increasing retrieval effectiveness. (3) They use a closed, centralized and typically small geographic data set, and cannot make use of multiple data sets.

In this paper a novel system is presented for GIR on the World-Wide Web. A wide and extendible set of elements (e.g. postal codes, geographic names) is used as geographic indices of the documents. Query evaluation is based on distance measures related to transportation (e.g. travel-time and travel-cost distances).

The proposed system integrates autonomous information systems available on the Internet: general purpose Internet search machines, geographic databases and transportation related information systems. The applied mediator based architecture ensures a loose integration of the information systems. Key issues are handling uncertain data, and achieving a reasonable performance.

1 Introduction

Giving the best possible support to discover relevant pieces of data to meet information needs is one of the most important goals and challenges in the development of information systems. Currently, the importance of unstructured and semi-stuctured textual documents is growing, due to the rapid expansion of the World-Wide Web (WWW) and the widespread use of computers for document creation and management. Information Retrieval (IR) is focusing on finding data objects (DO), typically text documents, relevant to information needs in various situations [1].

The relevancy of a piece of data to a particular information need is usually determined by several factors. In many fields, ranging from leisure-time activities to commerce, from public administration to environmental protection, the geographic reference of the DOs and that of the information need effect relevancy fundamentally. One data item may be relevant to the information need, while another not, although the only difference between those is their geographic reference. Increasing mobility and free-time activities make this issue especially timely.

Geographic Information Retrieval (GIR) [2] is concerned with the geographic aspect of relevancy. It concentrates on how to determine what pieces of data are geographically relevant to certain information needs. Similarly to general IR, GIR focuses primarily on unstructured and semi-structured textual documents[1]. However, since geography is concerned with *spatial* objects on *Earth*, a system for GIR has to handle different space models and large geographic databases. Present techniques and tools for general IR are generally inappropriate for GIR.

1.1 Outline

In Section 2 of this paper, the cornerstones are addressed: Information Retrieval and formal models of spatial data. In Section 3, current approaches are presented, and their limitations are summarized. In Section 4, the new approach is described. In Section 5, a summary is given, and the status of the work is described.

[1] The expression "GIR" is often used in conjunction with structured geographic data. This word usage in inappropriate and misleading. In analogy with the expression "database management", "geographic database management" should be used in this case.

2 Cornerstones

2.1 Information Retrieval

Data retrieval and *information retrieval* [2] differ in two ways. First, while data retrieval is concerned with structured data, information retrieval is mostly concerned with unstructured or semi-structured, textual documents. Second, data retrieval concentrates on the technical side (i.e. which data fulfill query criteria), whereas information retrieval is more concerned with what is relevant to the information need.

2.2 Formalisms to Represent Spatial Structures

Methods and systems for handling spatial data use various formalisms to represent properties of spatial structures. These formalisms are called geometries [3]. For our purpose, important geometries are the Euclidean, the hierarchical set-based, the network and the topological space models (see Fig. 1 for examples).

In the different space models different relations between spatial objects can be interpreted. Examples for the possible relations are closeness in the Euclidean, containment in the hierarchical set-based and neighborhood relation in the topological space models. The space model applied by a GIR system determines which relations can be handled and thus strongly effects the potential retrieval effectiveness.

Another approach focuses on travel time distances, i.e. how long it takes to get from one place to another using a transportation network. Fig. 2 shows the one-hour travel time neighborhood of a city. The shaded areas can be accessed within one hour from the city center with a combination of public transportation and walking.

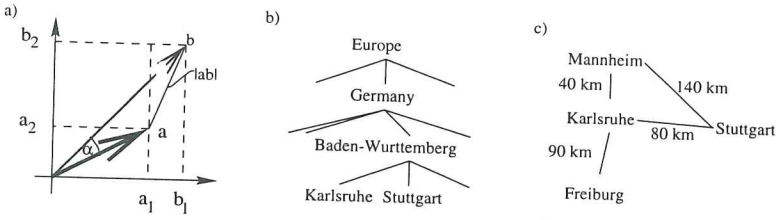

Figure 1: Examples for data in some space models: a) Euclidean space model; b) Hierarchical set-based space model; c) Network space model

3 Related Work

3.1 Geographic Indices in Library Catalogues and Metadatabases

In traditional and digital libraries catalogues and metadatabases support geographic indices. Prominent representatives are the MARC (Machine Readable Catalogue) standards [5], the locator record specification of the Government Information Locator Service (GILS) [6] and the Dublin Core metadata element set specification [7].

Geographic indices in the above and similar standards and specifications have two major forms: textual indices and geometric indices. Textual geographic indices contain words: geographic place names and occasionally modifiers. They have two forms: they are either based on standardized geographic thesauri or are non-standardized, free-text geographic descriptions. Geometric indices contain a set of coordinate values in a specified coordinate system. Geometric indices are either bounding boxes or more detailed polygonal descriptions.

3.2 Internet Search Machines

Internet search machines (e.g. AltaVista [8], Infoseek [9]) support primarily free-text search, normally used for general, topical IR. This also allows the finding of geographic place names in the DOs. Thus some, even if very limited, GIR is possible.

The Internet domain name conventions open another possibility for GIR. The

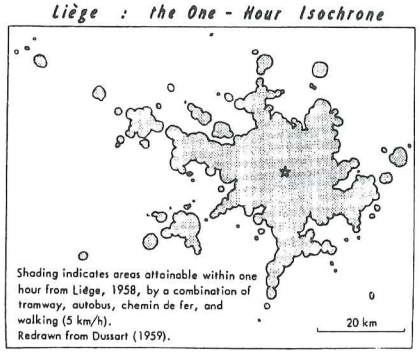

Figure 2: Travel time neighborhood of Liege/France (from [4])

last part of an Internet domain name identifies the country of the domain. By making a restriction on the domain name, which is supported by practically all search machines, the search can be limited to one country. The HotBot search machine [10] supports geographic conditions referring to continents. This is based, beside the domain name conventions, on a small geographic database.

3.3 Automatic Geographic Indexing of Text Documents

A method for automatic geographic indexing of text documents is presented in [11] and [2]. The method is based on a large geographic gazetteer, containing geographic names and corresponding geometric representations.

The automatic geographic indexing of a document is based on the recognition of geographic place names in the text, on the retrieval of the corresponding geometric representations from the gazetteer, and on combining the individual geometric representations. The resulting geographic index of the text document is a *fuzzy footprint.*

3.4 Limitations of Previous Approaches

The major limitations of previous approaches are as follows.

- They are concerned with the geographic indexing of the DOs, while query evaluation is neglected. This implies a straightforward comparison of geographic indices and geographic query conditions. Consequently, only exact but no approximate matches are found.
- They make limited use of formal space models. Consequently, they cannot make use of relations defined in the different space models (e.g. part-of, neighbor-of) to improve retrieval effectiveness.
- They use a closed, centralized and typically small geographic data set. They cannot make use of multiple data sets that are autonomously collected and maintained. This is not the appropriate approach for geographic data sets, which are typically large, expensive, and of limited coverage and detail.

4 Approach

4.1 Information Retrieval Aspect

In text documents, geographic reference is carried by several kinds of data. Some of those refer to larger geographic areas and some to fairly specific locations. GIR has to make use of all these pieces of data. Examples are postal codes, telephone area codes, addresses and car registration numbers. In certain application areas further elements can be added to the list. For instance, in public administration, names of local government members carry a geographic meaning, too. The presented approach makes use of a wide range of such data, and can be readily extended with additional data sets.

The geographic relevance of a DO to a query is determined by some *distance* between the geographic references of the two. In many application areas, distances are determined by traveling, transportation or telecommunication possibilities. The presented approach is based on distances related to transportation, e.g. travel-time and travel-cost distances.

The proposed system allows GIR on the WWW. An example query may sound as follows: "Find the WWW pages on the Internet describing objects accessible from *Karlsruhe* by *public transport* in less than *two hours*!"

4.1.1 Handling of Uncertainties

There are several factors leading to uncertainties in query processing. The query result can be permeated with nonspecific type uncertainty for the following reasons:

- the geographic location of a DO may not be determined exactly, because (1) the geographic reference contained in the DO is not specific enough, e.g. only a telephone number with area code, but no postal address is given, or (2) the data necessary to interpret the (exact) geographic reference of a DO is missing, e.g. the DO contains full postal address, but there is no map available for the area.
- the distance between two locations cannot be measured precisely, because (1) the travel medium is not specified (e.g. very high speed train or local train), (2) the time of the travel is not specified (e.g. Sunday morning, which is off-peak, or Friday noon, which is peak time) or (3) disturbances may occur.

From the user's point of view, the handling of uncertainties is twofold. First, query execution can be tuned between high precision (high portion of the retrieved DOs are real matches) and high recall (high portion of the matching DOs are retrieved) modes. Second, query answers are annotated with the sizes and the sources of the associated uncertainties.

4.2 Realization Aspect

4.2.1 General Architecture

The system for GIR on the WWW is built up by integrating distributed, autonomous information systems available on the Internet. Information systems of the following three groups are integrated:

- Geographic and relevant databases (e.g. postal code databases, telephone area code databases, geographic gazetteers);
- Information systems related to transportation and traveling possibilities (e.g. car route planning and public transportation timetable services);
- General purpose Internet search machines.

The integration of the autonomous information systems is loose. I.e. new services can be easily added to the overall system, all pieces of data are utilized and the overall system is still operative, even if some information systems are not available.

The system architecture (Fig. 3) corresponds to the mediator-based model, originally suggested in [12]. Beside the remote information systems of the above mentioned three groups, the major components are the wrappers, the mediator and the user interface. There is one entry point to the system, the user interface. It supports information need oriented, high level query formulations. Details, such as what the underlying information systems are or how they are accessed, remain hidden from the user.

A similar architecture is used in information integration and digital library projects. However, these projects are concerned with data from relational databases [13] or are based on a hierarchical object model [14], and do not support the handling of uncertainties and of spatial data models.

4.2.2 Mediator

The mediator receives high-level query requests from the user interface. Leaning on its internal knowledge on the available information systems, it creates a query execution plan. The plan contains data on which remote information systems have to be queried, what parameter settings for these subqueries have to be used, and in which order the subqueries have to be executed. There are typically several alternative plans. The mediator estimates the execution costs associated with each of those, and chooses the plan with the lowest cost.

The second task of the mediator is to control query execution. According to the prepared query plan, it sends subquery requests to the remote information systems, and accepts the partial answers. Dynamic plan modification may also be necessary, as the behavior of the autonomous remote information systems may change.

The third task is to assemble the partial answers returned from the remote information systems into the main result list. Different information systems may use different space models. Therefore, space models have to be handled in a flexible way. Relations of the different space models must be combined, so that all pieces of information can be used. Also, the merging of uncertain information has to be handled.

The mediator contains data on the remote information systems. This data is stored in the form of declarative descriptions, which is required for the loose integration. When a new information system becomes available, solely its description has to be added to the mediator's repository.

The mediator also requires knowledge on the relations between spatial objects. Both generally valid, static knowledge and dynamic knowledge used in a particular

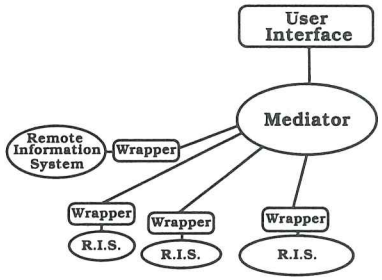

Figure 3: Top-level architecture

Set-based	`(1111 is-part-of 111?)`
Euclidean	`(1111 is-part-of 111? coverage 10%)`
Topological	`(1111 is-neighbor-of 1112)`

Figure 4: Relations of spatial objects in different space models: an example of postal codes

query are required. The knowledge is stored in rules, allowing for the flexible handling of various space models (see Fig. 4 for an example). Two types of such rules are used: general or class-level rules and specific or object-level rules.

4.2.3 Wrappers

Each remote information system is integrated into the overall system with a separate wrapper. The primary responsibility of the wrapper is to hide the technological details of the access to the remote information system. Easy construction of wrappers for new information systems is a precondition for the required loose integration. The remote information systems typically have a WWW interface, accessible with the HTT protocol. On this topic much work has been done recently (e.g. [15] [16]), which offer satisfactory approaches for our purpose.

The second task of the wrapper is to extend the services of the remote information system. As an example, a public transportation timetable service can tell how long the journey from Karlsruhe to Ettlingen takes on Monday morning, at 10 am. In GIR, the reasonable query is often not all that specific. Useful issues are how long the journey on average takes, what the range and the distribution of the required time is, and what the influencing factors are. The wrapper supplies these additional pieces of information, which are not directly available in the remote information system.

The additional information can be supplied in two ways. The wrapper processes all queries to the information system, and calculates some statistics, which can be used to supply the additional information. It may happen, however, that the available statistics are insufficient. In these cases, the wrapper performs multiple queries to the underlying information system, and calculates the results from those.

The third task of the wrapper is performance improvement. In GIR, the transportation information systems may become overloaded. There are two main reasons for this. First, the queries they are prepared for, and those necessary for GIR are

each others' reverse. (E.g. in a timetable the input parameters are two locations, and the output parameter is travel time. In GIR, the input parameters are one location and the travel time, and the output parameter is the list of locations.) Second, these information systems usually perform computationally intensive functions (e.g. route optimization).

The goal is to reduce the number of queries to those remote information systems. For this purpose, the wrapper builds up a draft model of the database of the remote information system. The wrapper thus becomes active, and is able to answer some of the queries without having to contact the remote information system. In the case of travel and transportation information systems, the draft model can be e.g. the network of high speed train lines or motorways. Making the wrappers active has one more advantage: even if the underlying remote information system is not available, e.g. due to overloading, the active wrapper still delivers some approximate data.

5 Summary, Status

In this paper, current approaches to GIR in traditional and digital libraries have been analyzed. Their major shortcomings have been identified as follows: focusing on indexing; making limited use of space models; using fixed, small geographic databases.

A new framework for general purpose GIR on the WWW have been presented. Concerning the IR aspect, its main features are handling a wide range of geographic indices and using distances related to traveling and transportation for query evaluation. The system is based on autonomous information systems, which are loosely integrated. The main challenges are to combine the space models in a flexible way, to improve performance, and to handle uncertainties.

The status of the work is as follows. Test queries have been executed by hand to check the feasibility of the approach. Universal WWW search machines have been evaluated concerning their relevant features (e.g. query language: support for joker characters, maximum number of conditions in one query; metadata accompanying query results: number of hits, number of partial hits). A universal wrapper, appropriate for all three kinds of information systems which are to be integrated, is currently being developed based on the approach described in [16]. A wrapper with

semantic extensions for transportation information systems is also being developed. It is capable of generalizing answers and is active, i.e. it contains the network of the fast transportation lines. Java was chosen as the major development language, because of its platform-independence and support for transferable applets, network communication and database access. The Python programming language [17] is currently being evaluated for its usability in extracting data from HTML documents.

Future work is planned on the data model and query language supporting uncertainties and approximate matches.

References

[1] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[2] Ray R. Larson. Geographic information retrieval and spatial browsing. In *Geographic Information Systems and Libraries.* Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, 1996.

[3] M. Worboys. *GIS: A Computing Perspective.* Taylor & Francis, 1995.

[4] W. Tobler. Non-isotropic geographic modelling, in three presentations on geographical analysis and modelling. Technical report, National Center for Geographic Information and Analysis, University of California at Santa Barbara, CA, USA., 1993. 93-1.

[5] Ellen Gredley and Alan Hopkinson. *Exchanging Bibliographic Data, MARC and other international formats.* Canadian Library Association, The Library Association, American Library Association, Ottawa, London, Chicago, 1990.

[6] William E. Moen. The government information locator service: Discovering, identifying, and accessing spatial data. In *Geographic Information Systems and Libraries.* Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, 1996.

[7] Stuart Weibel and Eric Miller. The Dublin Core Metadata Element Set Home Page, 1997. available at http://www.oclc.org:5046/research/dublin_core/.

[8] Digital Equipment Corporation. AltaVista Home Page, 1997. available at http://altavista.digital.com/.

[9] Infoseek Corporation. Infoseek Home Page, 1997. available at http://www.infoseek.com/.

[10] Wired Digital, Inc. HotBot Home Page, 1997. available at http://www.hotbot.com/.

[11] Allison Gyle Woodruff and Christian Plaunt. GIPSY: Automated geographic indexing of text documents. *Journal of the American Society for Information Science*, 45(9):645–655, 1994.

[12] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[13] Oliver M. Duschka and Michael R. Genesereth. Infomaster — An Information Integration Tool. In *Proceedings of International Workshop on Intelligent Information Integration, Freiburg, Germany*, 1997.

[14] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and Jennifer Widom. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering, Stanford, California*, pages 61–64, 1995.

[15] Naveen Ashish and Craig Knoblock. Wrapper Generation for Semi-structured Internet Sources. In *Proceedings of the Workshop on Management of Semistructured Data, Tucson, Arizona*, May 1997.

[16] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Proceedings of the Workshop on Management of Semistructured Data, Tucson, Arizona*, May 1997.

[17] Corporation for National Research Initiatives. The Python language home page, 1998. available at http://www.python.org.

A Knowledge Base Server and Its Application to Web Searching

Claudia Oliveira[1], Júlio Cezar Duarte[1]
Marcel Augustus[1], Marcelo Sant'Anna[2]

[1]Instituto Militar de Engenharia, e-mail: {cmaria,duarte,augustus}@ime.eb.br
[2]Pontifícia Universidade Católica do Rio de Janeiro, e-mail: draco@puc-rio.br

Abstract

This work describes a tool specialized in knowledge processing, the WittyServer, and the suitable architectural framework of client-server application in which WittyServer can function. This tool is designed to serve applications which require, among other things, First Order Language knowledge bases with classical and non-classical reasoning, and formal language processing. The description of the system's capabilities and resources outlines the requirements for communication with other appplications. A case study is presented which motivates the use of the knowledge base server within the context of Web Searching.

1 Introduction

Mainly due to the ever-increasing adoption of Web technologies, inside and outside companies, demand for distributed applications has strongly increased during the last years.

Given this context, large amounts of data are spread over networks where textual, relational and object oriented database servers are the most sought after and available tools used by software engineers to build information systems.

This paper presents an on-going research work on the construction of a knowledge base server called WittyServer. This tool is targeted to be an important software component in distributed information systems, mainly when the cross-relation of complex amounts of knowledge is a major concern.

WittyServer is designed to be used as a server for applications which require, among other things, First Order Language (FOL) knowledge bases with classical and non-classical reasoning, formal language processing with grammars, and Logic Programming.

This work is presented as follows. In section 2 Witty is described, with particular focus on the structure of knowledge bases and the theorem prover. The description of the system's capabilities and resources outlines the requirements for communication

with other appplications. These requirements will be defined in section 3 when we describe WittyServer. In section 4 the application of the knowledge base server within the context of Web searching is explored.

2 Overview of Witty

The Witty environment is composed of a theorem prover with a knowledge base structure for FOL formulas and a configurable inference engine which supports the implementations of classical and non-classical reasoning mechanisms, together with a Post style grammar interpreter and a host programming language.

The environment, which was originally built to support the development of stand-alone applications such as expert systems, supports program editing, knowledge base editing, debugging and other user interface facilities.

2.1 Architecture

Here we give a brief overview of the Witty environment, which is fully described in [12]. Witty can be best described through an architecture composed by four levels of functionalities: the logic level, the grammar level, the interface level and the logic programming level.

Firstly, the logic level is a hierarchical structure of knowledge bases and a first order inference procedure. The knowledge base structure is a collection of knowledge bases (KBs) defined as pairs $< s, \Gamma >$, where s is a label and Γ is a multiset of clauses. The hierarchy of the KBs is given by the ordering relation underlying the KBs labels. By default, the labels are linearly ordered by creation time. Other types of ordering require extra programming.

The inference procedure is such that, if used as a classical prover it is equivalent to resolution but with the addition of the **Oracle** device it can implement a range of non-standard reasoning mechanisms [12].

Basically, an **Oracle** is a program in Witty's host language which extends resolution such that an alternative direction will be given to the proof-search whenever the end of an unsuccessful path is reached. An oracle will seek to solve the problem outside the current setting, it will use extra data. If an oracle $op(L, R)$ is applied after a failed resolution step then L is the literal which failed to be resolved and R is the current resolvent. These arguments have been chosen on a purely experimental basis, i.e. there has yet to be a study of the theoretical implications of the arguments of the oracle program. Nevertheless, the unresolved literal is an obvious choice and the resolvent has been required in some researched applications. If *op* succeeds then the derivation proceeds; otherwise the derivation backtracks.

Secondly, the grammar level consists of an interpreter of Translation Grammars(TGs), which is a language designed to implement pattern matching in a knowledge processing environment. The language is built upon a small set of consistent concepts, inspired by Post Production Systems.

The Interface level is divided into two major modes of external data exchange: user interface and Web interface. The former a collection of visual objects which allow the user to communicate with the system whenever the system requires information or wants to provide information. They include dialog boxes, push buttons

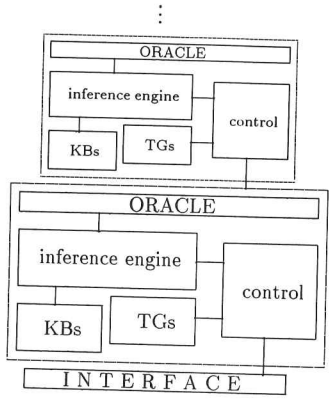

Figure 1: Witty archiecture

and various display facilities. The latter is a group of facilities to access Web nodes. The Web interface was implemented with **LibWWW**, a WWW library for C [1] available over the Internet. LibWWW is a general-purpose Web API written in C, which can be used as the code base for writing Web clients and robots. LibWWW provide an implementation of HTTP and other Internet protocols.

The fourth level is a program interpreter, the **Control**, which integrates the logic and grammar levels. This integration is promoted by a Logic Programming Language with a set of primitives, functionally grouped by level. Typically, the logic level is used via the primitives *prove, assert, newbase, delfact, delrule, lsbase, setoracle*; the grammar level is used via the primitives *newreg, assign, retrieve, loadgram, appgram*; the interface level is used via the primitives *input, prompt, ask, aqscon, choose*, for user interface; and *getURL* for Web interface. There are groups of common control primitives such as *cut, fail, loop* and the usual arithmetic primitives. The Control responds to **commands** which are sequence of primitives in Witty's host language.

3 The KB Server

Recently, Witty has been stripped of all user interface functionalities and has become WittyServer. Instead of offering an environment, WittyServer serves requests for storing formulas in knowledge bases, carrying out proofs from these bases and other specialized knowledge processing tasks.

The transition from user friendly environment to server application demanded a detailed study of Witty's functionalities and a review of its programming language. This study involved the conception of the client-server framework in which a knowldge base server would be most useful and also most reliable.

Some key changes allowed the insertion of the program into a client-server setup. These changes are related to the way in which the functionalities and resources of

[1] W3C sample code library is covered by the **MIT Copyright Statement**, and with acknowledgment to **CERN**.

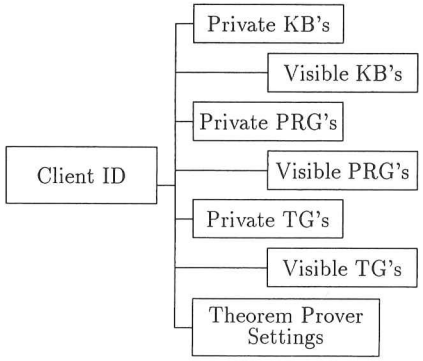

Figure 2: Client's Frame

WittyServer have to be shared by several client applications, and to the way in which these client applications will communicate with the server. A conceptual presentation for the WittyServer architecture is shown here, leaving networking implementation details out of the scope of this paper.

3.1 The Client's View of the Server

Upon connecting with the server, the client is given a **frame**, as seen in figure 2. A frame is like a directory of knowledge bases, programs, grammars and settings according to which the server will carry out the client's requests.

Initially the frame is empty and the theorem prover's settings are the default ones. As the client sends requests for loading KBs, PRGs and TGs, its list of Private KBs, Private PRGs and Private TGs are appended, respectively. The setting changing commands (*setoracle, setdepth*, etc) are used to update the theorem prover's settings for that particular client. Additionally, the client might request access to a set of public KBs (PRGs and TGs), in which case this information is recorded in its list of Visible KBs (PRGs and TGs).

The conficts arising from this Private/Public visibility are handled in much the same way as variable scope determination in Pascal programs.

Although the use of client-server point-to-point connections is common, the Client ID tag in the frame and in all the exchanged messages provides a desirable degree of independence from the implementation choices.

There are two possible scenarios where a knowledge base server could be used. Firstly, as a totally general-purpose KB server in which all the domain specific knowledge would be private to each client and only absolutely general knowledge would be shared. This approach implies that a client would have to provide all its KBs, programs and inference engine settings. Therefore, it requires a great deal of effort by each client to configure its view of the server with the extra disadvantage of overloading the server with data, in the case where several clients have complex setups.

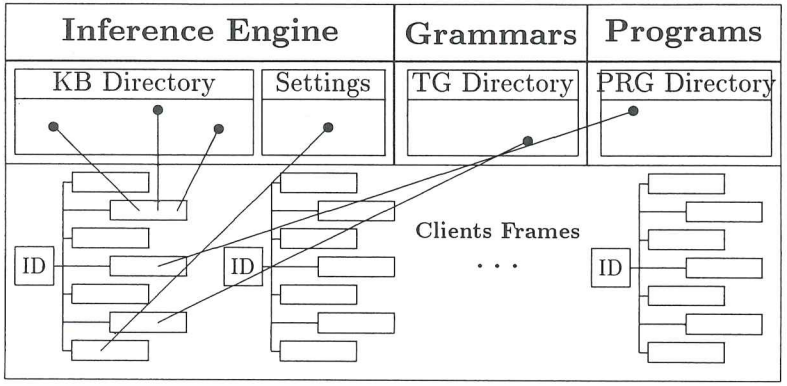

Figure 3: Domain-specific KB Server

In the second scenario, which seems the most sensible, the KB server is domain-specific. The server contains an Expert System for a certain domain and publishes the predicates, procedures and grammars names and descriptions. Each client can inspect the pre-defined resources and build its specific KBs (procedures and grammars), within the set of meaningful predicate symbols available, requesting access to public KBs (procedures and grammars).

Besides efficiency and economy, the second approach has considerable maintenance advantages. The public domain-specific knowledge can be updated for all clients at the same time. If there is a policy of reviewing and validating the public KBs to ensure consistency and correctness, then their maintenance can actually be distributed. A schematic view of the client frames in relation to a domain-specific KB server is shown in figure 3.

3.2 Communication Protocol

The implementation of the communication protocol has been designed considering a client-server architecture using callbacks. Client applications must be developed with a message handling module designed, not to only send requests, but also to monitor possible incoming messages from the server.

Basically, the client will be sending the server messages containing a command (sequence of primitives in Witty's host language) to be executed. If, in the course of execution, the need arises for the server to request information from a client, then a message is generated to the client containing the smallest possible description of the required information. The client may then treat this message and answer the server with the appropriately formated response. Message types can summarized as follows.

1. Client-Server The general format of communication messages from client to server is: <message size>,<client ID>,<message string>. Usually, the message

string is any valid command in the host languages. If the client's message is sent in response to a server's callback then the contents of the message string may vary accordingly.

2. Server-Client The general format of communication messages from server to client is: <message size>,<client ID>,<message type>,<message string>,<list of parameters>.

The message size and the client ID are always sent to maintain the generality of the message exchange, consisting of the message header. The full description of the message types, arguments and the host language primitives which generate the massages is given in Witty's Help Files.

The messages received by the client can be processed in any way: in general they are meant to be displayed or prompted to the end user, but there are cases in which the client application will provide a response without consulting the end user. Messages can also remain untreated, as long as they are removed from the message queue and an acknowledgement is sent, in the case of modal interaction.

3. Other Considerations Briefly, in the implementation of WittyServer several choices had to be made. Firstly, the choice of protocol for interoperability was TCP/IP because it has become a standard. Sockets were used as the interface for programming network applications in order to provide independence of the application from the transport layer. As far as memory sharing is concerned a multithread scheme has been chosen.

4 Web searching

It is redundant to state the impact of the World-Wide Web (WWW) as a global source of information in all areas of human interest, ranging from commerce to science. The potential for exchange and sharing is not yet matched by the ability to actually access and retrieve specific information of interest to the user.

As anyone who uses the WWW knows web searching is a particularly challenging task, due to the enormous volume of information that can be provided to the user making a query. Even when that query is well specified, a large amount of completely irrelevant information may be returned.

In [3] there is a very interesting classification of the various approaches to the development of so-called Global Information Management Systems (GIMSs). The main goal of such systems is to provide a framework to integrate heterogeneous information sources into a common domain model.

The first generation GIMSs would be the popular search tools, that the ordinary individual user has been employing in order to access WWW pages of interest, with various degrees of user-friendliness and effectiveness: Alta Vista, Lycos, Excite, to mention just a few. These are, in fact, interfaces to a massive index database, periodically maintained. The occurrence of words and expressions, with limited number of combinators, in a page yields an answer page of links.

The attempts at overcoming the limitations of those search tools have adopted, roughly speaking, two approaches: Database and Knowledge Representation based. The former aims to build a conceptual schema of the information domain together

with procedures to retrieve information, based on the schema. The latter relies on methods for dynamically accessing the information sources.

Among the Database proposals, two main streams can be found:

development of query languages: most notably W3QS [7], WebLog [8] and WebSQL [10]. These types of query languages make use of whatever structure the WWW provide. The user of these languages is not the WWW end user but an application developer.

development of wrappers and mediators for heterogeneous data-sources. The different types of data-sources in the WWW, structured or not, are ssumed to have a descriptor. For each type of data-source thus described a wrapper is built which will suitably translate or "interpret" queries. These systems are in line with the DARPA Intelligent Integration of Information (I^3) research program.

The fundamental difficulty encountered in all the approaches is the lack of a rigid structure of the Web, as a database, on which client systems can rely on to manipulate documents. At the document level, the structure is also lacking, notwithstanding the metadata available at a HTML page header and other scattered metadata.

Knowledge-based GIMSs use knowledge representation techniques (typically, description logics) to deal with information source representation, query processing and data acquisition. Noteworthy examples of this class of GIMSs are Internet Softbot [5], SIMS [1] and Information Manifold [9] among others.

4.1 Witty-WWW-Query-Server (W4QServer)

In what follows we describe the domain-specific query processing unit of a knowledge-based GIMS. It takes full advantage of WittyServer's KB structures and frames, integrating knowledge bases where general knowledge within a target domain is publically available and other preferences can be tailored to the individual and stored in private KBs. There is also the possibility for extending the maintenance of the public settings to all the users in a certain community.

The dynamics of a user interaction with W4QServer is shown in figure 4. From the HTML browser, the user simply inputs keywords. There are 7 sub-tasks involved in the query processing task.

Query enrichment: the addition of the following items to the initial keywords provided by the user: a) synonyms, hypernyms and hyponyms inside the fields the user wishes to search (e.g. in the case of "agents", the addition of "distributed", "computational", "artificial intelligence" and so on) obtained from the ontology and terminology KBs; b) language equivalents when the query is meant to cover Web documents in different languages (e.g. "agentes"); c) authors, since our test cases have been in the field of *research in computing*; d) prefered domains; e) prefered sites and servers; f) date ranges. Some of these additions are ineffective to some of the search engines used.

URL synthesis: several URLs are built from the enriched string of words, to comply with the syntax of the various employed search engines.

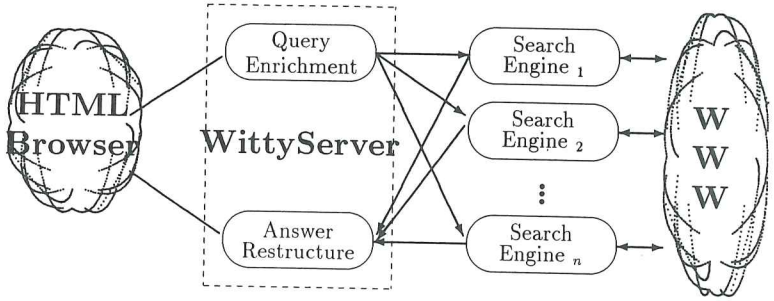

Figure 4: W4QServer Architecture

URL request: the URLs are posted to multiple search engines in parallel.

Obtaining answer pages: the answer pages are returned, in different response times, by the search engines.

Interpretation and restructuring of answer pages: a parsing of the answer pages is applied to the answer pages as they arrive in order to eliminate repetitions and reorder links. Further restrictions not supported by the search engines can also be applied here, to the initial lines of the link pages or to the page itself.

Generation of new answer pages: a new answer page is constructed and returned to the HTML browser as the final response to the user.

The Knowledge Bases are of three basic types and are structured as in figure 5.

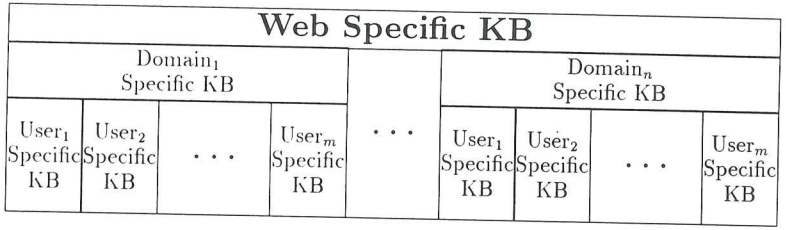

Figure 5: Knowledge Base Structure

1) User specific KBs, where the profile of the user is modelled with respect to her/his main interests in the targeted field. These private KBs can be dynamically maintained via learning from the history of the users queries. 2) Domain specific KBs, where the ontological and terminological model of the target domain is represented. These are public KBs and are centrally maintained. 3) Web specific KB, containing

knowledge about the structure of the Web and descriptors of the search engines used.

A number search tools with query mechanisms, which have a similar purpose of improving the results and end-user interfaces of the search engines, are available, such as MetaCrawler [11], InferenceFind [6] and WebTrec [13] (for a list of Web search tools see [4]). The main advantages of our approach are:

1. the structure of the KBs which classifies user profiles, domain models and Web models;
2. the knowledge bases are themselves viewable, giving the user a possible explanation for the results obtained;
3. the high degree of maintainability, to cope with the rapidly changing WWW.

5 Concluding Remarks

This paper describes a knowledge processing environment and its server version. The transition from one architecture to the other is demonstrated through the definition of the client's view of the server and the additional communication requirements.

The application W4QServer has been chosen from a group of WittyServer based applications which include a Knowledge Acquisition tool and an Agent-based architecture. This choice has been made, firstly because Web searching is currently a very challenging and active field of research. Secondly, we wish to motivate software developers and researchers in this area to apply the use of knowledge bases as part of their WWW projects.

Acknowledgements

Much of this work was developed while the authors was being supported by CNPq on a grant. During the same period, FAPERJ has contributed with funds for aquiring much valued computer equipament.

The authors wish to thank Prof. Roberto Lins de Carvalho who was the first to conceive Witty, and who has given a lot of support and advise in this project.

References

[1] Yigal Arens, Knoblock C., and Shen W. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 1–38, 1996.

[2] Sonia Bergamaschi and Sartori C. An approach for the extraction of information from heterogeneous sources of textual data. In *Proceedings of the 4th KRDB Workshop*, Athens, 1997.

[3] Tiziana Catarci, Iocchi L., Nardi D., and Santucci G. Conceptual views over the web. In *Proceedings of the 4th KRDB Workshop*, Athens, 1997.

[4] Search Engines. Http://www.geocites.com.rainforest/3922/search.htm.

[5] Oren Etzioni and Weld D. A softbot-based interface to the internet. *Communications of ACM*, 37(7), 1994.

[6] InferenceFind. Http://m5.inference.com/ifind/ifind.cgi.

[7] David Konopnicki and Shmueli O. W3qs: a query system for the world wide web. In *Proceedings of VLDB'95*, pages 54–65, 1995.

[8] Laks Lakshmanan, Sadri F., and Subramanian I. A declarative language for querying and restructuring the web. In *Proceedings of the 6th International Workshop on Research Issue in Data Engineering*, 1996.

[9] Alon Y. Levy, Rajaraman A., and Ordille J. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference*, Bombay, 1996.

[10] Alberto Mendelzon, Mihaila G., and Milo T. Querying the world wide web. In *Proceedings of PDIS'96*, 1996.

[11] MetaCrawler. Http://metacrawler.cs.washington.edu.

[12] Claudia Oliveira. *An Architecture for Labelled Theorem Proving.* PhD thesis, University of London - Imperial College, 1995.

[13] WebTrec. Http://www.jwsg.com.