METAMODELS IN GENERATION OF UML USING NLI-BASED DIALOG

L. Kovács[*], G. Kovásznai[**] and G. Kusper[**]

[*] University of Miskolc/Department of Information Technology, Miskolc, Hungary
[**] Eszterházy Károly College/Institute of Information Technology, Eger, Hungary
kovacs@iit.uni-miskolc.hu,

Abstract— **The current trends in software engineering aim at narrowing the semantic and syntactic gap between the representation formalism of humans and of computers. This paper gives an overview on the application of natural language interface in software engineering applications and describes the structure of a NLI-based UML model developer module. An extended UML Kernel metamodel is presented for the NLI dialog interface and UML generation.**

I. INTRODUCTION

The activity of programming means the transformation of internal human knowledge into binary code sequence. Due to the huge differences in the representation forms and the complexity of the problems, this is usually only an approximation. The main difficulties of the programming process are the followings:

- ambiguous and fuzzy concepts of humans
- ambiguous and fuzzy language terms of humans
- terms in different contexts have different meaning
- different people have different context background
- our thinking is based on approximation, while computers are crisp machines
- a high level concept can be described only with a huge amount of base elements

The huge gap between humans and processors causes high cost in programming. Thus, the trends in software engineering aim at narrowing the semantic and syntactic gap between the representation formalism of humans and of computers (easy to write code, easy to understand) make the code and model reusable (easy to rewrite it, easy to port it to other systems, flexibility).

In order to achieve these goals, there is a steady evolution in software engineering techniques. One the development areas is the field of programming languages, new and new programming paradigms are developed in the recent decades. A *programming paradigm* is a way of conceptualizing describing how tasks that are to be carried out on a computer should be structured and organized. In [11], the programming paradigm covers three different aspects: code execution, software decomposition and models of computation. From the viewpoint of semantic transformation, the main component is the code execution aspect that describes the relationship between program code and the binary code regarding the formalism and execution method. The main execution models are [11]:

- algorithmic mode: the program describes the control flow, the basic steps of execution
- declarative mode: it describes how the output data should depend on the input data,
- rule-based mode: based on stored rules, new facts are implied from existing facts.

There are different development stages in each subcategories. The first algorithmic languages used a low level control structures using *imperative* code style. The subsequent versions are based on procedural or structured approach in order to cope with the huge amount of base code instructions. In this case, the programmer can refer to higher compound units as elements in the source code [C, Fortran,..]. In the *declarative* languages the programmers should not care with the execution details. The programmer describes only the required property of the output, and the language processor will determine the required instruction sequence for the problem. The SQL language is called usually a declarative language. *Declarative* code is based on the concepts of function, relation, and logic as developed in mathematics. The relational calculus is also a good example of the declarative language. In the rule based systems, the developers describe the relationship between the events and responses. The logic programming languages like DATALOG is member of this family.

II. NLI AS PROGRAMMING LANGUAGE

There were several efforts in the last decades to use a different language for program code than the standard programming languages. The first attempts are investigating the usage of *logic languages* as it is very near to the control structure of programming languages. In[3], different logic-based executable agent specification languages were analyzed and compared. These logic languages can be used for modeling and prototyping. The main benefit of logic languages is the strict and rigid interpretation, thus the conversion into programming code is a strait forward process. The comparison of the expression power in logic languages and natural languages are investigated among others in [15]. There are different approaches in this topic, some argue [12] that First Order Predicate Logic is suitable to describe all language synthoms while others prefer to use higher order logic [13].

The experiences show that the NLI interface in SE like in DBMS can be used only for some domain problems and not in general. Despite of pessimistic forecasts, some experts work on development of restricted NLI modules

for SE tasks, there are some early approaches to use NLI in code generation. For generation of new objects and for setting the attribute values of existing objects a hierarchical NLI interface was developed in [8]. The hierarchical structure means here that the user composes the command traversing a rule tree top-down.

The efforts to combine NLI and programming languages belong to the umbrella term *natural programming* to make programming languages and environments easier to learn, more effective, and less error prone. It aims for language and environment to work the way that nonprogrammers except. The Natural Programming Project[2] is taking a human-centered approach, first studying how people perform their tasks and then designing languages and environments around people's natural tendencies. According to the studies, one of the most effective tools in natural programming is the application of a NLI interface. On other hand, it should be clear, that the NLI interface merits the most profit if it is coupled with an extended metadata, ontology database. This database provides the concept dictionary needed in the interpretation of the NLI sentences and it provides the knowledge base to support the code generation process.

Another domain specific application is the Inform programming language. Inform is a design system for interactive fiction, a new medium for writers which began with adventure games in the late 1970s and is now used for everything from literary narrative fiction through to plotless conceptual art, and plenty more adventure games too. The story description is given with a CLE (*Controlled English Language*). It can process only certain forms of sentence typical in given situations. Inform source text may look like "natural language", the language we find natural among ourselves, but it has several limitations. Some sentence structures like "something which is carried by the player" are understood while other sentences with the same meaning like "something which the player carries" can not interpreted. The Inform system has a limited knowledge base incorporated and can infer from the sentence "Peter wears a hat" that Peter is a person.

III. CONCEPTUAL SCHEMA AS INTERMEDIATE REPRESENTATION

For mapping NL input to UML diagrams, an intermediate stage is needed, founded in linguistic information, but with certain conceptual knowledge. In this section, we give an overview on state-of-the-art conceptual schemas, with considering the existing systems employing these schemas.

A few systems employ purely textual conceptual schemas. The system named *Requirement Elicitor* [1] normalizes the NL input into a sequence of textual constructs, of 4 types: "conditional", "iteration", "concurrency", and "synchronization". Let us note that this normalization is not fully automatic. A module called *NL-OOML* maps the content of the normalized input to an OO model, by following simple translation rules.

Unfortunately, the nature of the resulted UML diagrams is not specified by the authors.

As another textual conceptual schema, the *Two-Level Grammar* (TLG) can be employed, as proposed by [2]. A TLG consists of two context-free grammars; one for defining type domains, and one for defining functions operating on those domains. These grammars may be defined in the context of a class, thus TLG can be considered as an OO requirement specification language with an NL style. TLGs can be transformed to *VDM++* [10], an OO specification language, which has a tool support for analysis and code generation. Hence, a VDM++ specification can be further transformed to Java or to UML Class Diagram.

A large group of systems employs semantic networks for representing conceptual knowledge. A fine example is the *CM-Builder* [3] CASE tool, which attaches semantic knowledge to the syntactic tree emitted by the lexical preprocessor module. The semantic knowledge is represented in a simple predicate-argument structure, which is still textual, but can be considered as a semantic network. Predicate names can be nouns or non-copular verbs, and the arguments can be references to predicates or other features (e.g., tense, numbers, etc.) CM-Builder transforms NL input to an UML Class Diagram.

NL-OOPS [4] is another NL-based CASE tool employing semantic networks. NL-OOPS was founded on *LOLITA* (Large-scale Object-based Language Interactor, Translator and Analyser), which is in fact among the largest implemented NL processing systems. NL-OOPS implements algorithms for extracting classes and associations from semantic networks, hence, can emit UML Class Diagrams.

The *conceptual graph* (CG) [11] is widely employed as a semantic representation. A CG consists of concepts and conceptual relations as nodes, and arcs connecting the concepts with the relations. Furthermore, CGs can be nested with each other, in order to represent contexts. In connection with NL-based CASE tools, we are to mention *ASPIN* (Automatic Specification Interpreter) [5], which employs CGs in order to describe the behavior of a control system. The resulting CGs are joined into a so-called Process Model Graph, which is actually not a UML diagram.

Also a few improved variants of CGs are known. *Pre-conceptual Schemas* (PS) [6] intend to overcome a few drawbacks of CGs, like redundancy and ambiguity, as well as incapability for representing dynamic properties. Furthermore, PSs were created with the purpose of supporting the main features of the mapping to UML. Therefore, PS introduces new types of nodes (e.g. "dynamic relationship", "conditional") and arcs (e.g., "implication"). However the automatic obtaining of a PS from NL input is not yet solved, a CASE tool called the *UNC-Diagrammer* [7] can transform PSs to three types of UML diagram: Class Diagram, Communication Diagram, and State Machine Diagram.

Another improved variant of CGs is the *Extended Conceptual Graph* (ECG) [8], which is a predicate-centered schema language. Predicate nodes in ECGs are

similar to relationship nodes in CGs. ECGs employ five types of concept nodes, making the representation able to distinguish between classes and instances. New arcs are also introduced (e.g., an "isA" arc), as well as containers. It is not known whether ECGs support the obtaining of UML diagrams.

The *Bubble Lexikon Architecture* (BLA) [12] employs even more types of relations (e.g., "isA", "property", "function", "param" etc.), which can be additionally correlated with each other, like in the case of AND/OR graphs. BLA's name originates in the dynamically created and growing context "bubble" that can be attached to concept nodes. Considering the elements of BLA, it seems realizable to obtain a UML Class Diagram from a BLA diagram.

A promising graphical conceptual schema is the *Graphical Natural Language* (GNL) [9], which, in contrast with CGs, makes concepts dynamic, by letting concepts and relations participate in other relations. Last but not least, GNL is more compact as compared to CGs, as a consequence of the reducing of node types and the avoiding of redundancy. As a consequence of dynamism, one GNL diagram can be built for the entire text, as opposed to CGs. From the NL input, at first, a tabular representation is built, including three columns basically, as it is originates from the Subject-Predicate-Object structure. The tabular representation supports text analysis and verification, and employs heuristics to fill the empty cells. Then, the triplets in the tabular representation are transformed to concept (nodes) and relations (arcs) in the GNL diagram. Several types of nodes and arcs are supported, as well as the composing of them. From a GNL diagram, an UML Class Diagram can be obtained, as well as a so-called Use Case Path Diagram, which is actually not an UML diagram, but can be used as an intermediate form in order to obtain Sequence Message Charts or Use Case Diagrams.

IV.   METAMODELS IN GENERATION OF UML

The goal of the design phase is to develop an UML model of the problem domain. The structure of UML model is built up during a NLI-based conversion session between the developer and the application. The required structure of UML models is given in the form of metadata information. Our approach is based on the metadata description given in the MOF standard [MOF1]. The MOF model is a platform independent metadata framework. The MOF model is strongly related to the industry standard UML model and it defines a four layers metamodel architecture At the top layer, called the M3 layer, a meta-meta model can be found. This M3-model is the language used by MOF to build metamodels, called M2-models. These M2-models describe elements of the M1-layer, and thus M1-models. The last layer is the M0-layer or data layer. It is used to describe real-world objects. In this four-layered approach, the MOF 2.0 meta-meta model is located at the top level. The description of the UML model is at the third layer and the concrete UML models belong to the second layer. As a starting point, the UML Kernel (see Fig 1) metamodel is used to describe the ontology of UML models.

As this ontology description is used as a base for UML model construction, some additional relationships should be introduced in the UML Kernel model. One of the additional required elements is the dependency relationship. During the model building process, some elements are depending on the existence of other elements. For example, a property element should belong to a class element, thus the class element should exists before the property is created. This dependency relationship implies an ordering relationship also between the objects.

Another requirement is the definition of the relationship to the NLI elements. As the definition of the UML elements will be given in NL sentences, the related linguistic terms should be associated with the UML concepts. The key linguistic parameters are
-    related terms (words, expressions)
-    grammatical roles
-    synonyms
-    sentence schemas

In order to manage the different types of relationships and the information requirements of the NLI module, the standard UML Kernel metamodel [MOF1] was extended with more detailed relationship descriptions and additional linguistic annotations. The simplified structure of the resulted UML Kernel metamodel iks presented in Fig 1.. In the Figure, the rectangles with solid border represents the entity elements and the rectangles with slashed lines denote the relationship elements.

In the model building phase, an UML model instance is generated that matches the modified UML Kernel metamodel. The generated UML model is represented as graph that initially consists only of the empty compulsory elements. For example, an UML model instance should contain minimum one class definition. These frames are initially empty. During the design interaction, the missing properties are given by the user. After completing an UML element, some other optional UML elements can be added to the model. Thus, the modeling process means a continuous extension of the initially minimal empty models with additional elements. The ordering of the model elements depends on the applied UML Kernel metamodel. Thus the model contains a
-    kernel part: the elements are complete defined and a
-    border part: optional and empty element frames, that are related to the kernel part
-    current element (the target of the conversion)

The UML Kernel model is decoded into an OWL ontology to provide a standard ontology format. The generated ontology contains the following classes:
    - CLASS: UML class element
    - PROPERTY: attributes of classes

- OPERATION: methods of classes
- ASSOCIATION: association relationship between two classes
- ISA: specialization relationship between two classes
- HASA: containment relationship between two classes
- DATAPART: the link between the class and its attributes
- METHODPART: the link between the class and its methods

Some main properties of the classes are given in the following list:
- LEFT_ITEM: link to a class from an ASSOCIATION element
- RIGHT_ITEM: link to a class from an ASSOCIATION element
- OWNER: link to a class from an HASA element
- PARENT: link to a superclass from an ISA element
- CHILD: link to a subclass from an ISA element
- DTYPE: : link to a date format to an ASSOCIATION element
- RIGHT_CARD: the cardinality of an ASSOCIATION link

The CHILD property denotes the subclass in a specialization relationship. The OWL description of the corresponding definition of property CHILD is given as an example fragment from the OWL definition:

```
<owl:ObjectProperty rdf:ID="child">
    <rdfs:domain rdf:resource="#Isa"/>
    <rdfs:range rdf:resource="#Class"/>
</owl:ObjectProperty>
```

In the dialog , the application tries to fill in the missing information into the border elements. Based on the associated sentence pattern, a question is given to the user. The question is related to the missing elements, like
Does the <p_class> have additional properties yet?
where the <p_class> symbol denotes the current class element. The generated ontology stores the required linguistic data in a separate annotation properties, like

SYNONYMS: (instance level annotation)
PATTERN (class level annotation)

The sentence pattern to explore new properties are given in the Spattern annotation element within the Property definition.

```
<owl:Class rdf:ID="Property">
    <Spattern
rdf:datatype="http://www.w3.org/2001/XMLSchema#stri
ng"
```

>*What data is needed about <Class>?*</Spattern>
    </owl:Class>

Based on these language metadata, the NLI component of the engine can generate questions and can interpret the answers.

Having an initial ontology graph, the system explores the border area using a dialog interface with the users. The structure of the dialog sentences are generated using the sentence pattern. Having a Greedy algorithm, the system extends the ontology graph with new elements one by one.

V. QUALITY OF GENERATED UML DIAGRAMS

We have surveyed the articles which target UML diagram generation by NLP techniques. Now we want to judge the quality of these diagrams from the view point of object oriented design (OOD). This field helps the architect to identify classes and their responsibilities from the problem description or requirement specification. Furthermore, OOD gives principles which make sure that the designed software architecture is easy to modify and its components (classes) are easy to reuse.
The following 11 principles are most established in the literature:

- SRP, Single Responsibility Principle: A class should have one, and only one, reason to change.
- OCP, Open Closed Principle: You should be able to extend a classes behavior, without modifying it [18].
- LSP, Liskov Substitution Principle: Derived classes must be substitutable for their base classes [17].
- DIP, Dependency Inversion Principle: Depend on abstractions, not on concretions.
- ISP, Interface Segregation Principle: Make fine grained interfaces that are client specific.
- REP, Release Reuse Equivalency Principle: The granule of reuse is the granule of release.
- CCP, Common Closure Principle: Classes that change together are packaged together.
- CRP: Common Reuse Principle: Classes that are used together are packaged together.
- ADP, Acyclic Dependencies Principle: The dependency graph of packages must have no cycles.
- SDP, Stable Dependencies Principle: Depend in the direction of stability.
- SAP, Stable Abstractions Principle: Abstractness increases with stability.

There are two more principles given in the GoF book (Design Patterns, Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides), which are also commonly agreed to be the basics of design patterns, which are building blocks of well designed software architectures.
The two GoF principles are:

- Program to an interface, not an implementation.
- Favor object composition over class inheritance.

It is difficult to judge how good the generated UML diagrams are. First we shall define what does "good" mean. Our definition in this article is following: We say that an UML diagram is good if and only if its class diagram obeys OCP and LSP principles.

The cited UML generator articles discuss in details how do they do the first step, i.e., how do they identify possible classes and methods. Unfortunately they do not give details whether they obey the OOD principles. Therefore, we wanted to test them by the circle/ellipse problem, but we could not find the implementation on the public side of the Internet. We believe that future UML generators should also take into account OOD principles and try to fulfill our definitions.

VI. CONCLUSION

The application of natural language interface for program code development is a current topic in SE technology. The paper focuses on NLI for UML schema generation as an important step in software engineering. The proposed method uses an extended MOF metamodel for dialogue generation. The linguistic analysis is based on multi-layer parsing technology. Using a Greedy algorithm, the system extends the ontology graph with new elements one by one.

REFERENCES

[1] G. S. Anandha Mala, G. V. Uma, "Automatic Construction of Object Oriented Design Models [UML Diagrams] from Natural Language Requirements Specification", *Lecture Notes in Artificial Intelligence*, vol. 4099, pp. 1155 – 1159, 2006.

[2] B. R. Bryant, B.-S. Lee: "Two-Level Grammar as an Object-Oriented Requirements Specification Language", *Proc. of the 35th Hawaii International Conference on System Sciences*, pp. 3627-3636, 2002.

[3] H.M. Harmain, R. Gaizauskas, "CM-Builder: An Automated NL-based CASE Tool", *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, pp. 45-53, 2000.

[4] L. Mich, "NL-OOPS: From Natural Natural Language to Object Oriented Requirements using the Natural Language Processing System LOLITA", *Journal of Natural Language Engineering*, vol. 2, no. 2. pp. 161-187, 1996.

[5] W. Cyre, "A requirements sublanguage for automated analysis", *International Journal of Intelligent Systems*, vol. 10, no. 7, pp. 665-689, 1995.

[6] C. Zapata, F. Arango, A. Gelbukh, "Pre-conceptual Schema: a UML Isomorphism for Automatically Obtaining UML Conceptual Schemas", In: Advances in Computer Science and Engineering, *J. Research in Computing Science*, vol. 19, pp. 3–14, 2006.

[7] C. Zapata, A. Gelbukh, F. Arango, "A Novel CASE Tool based on Pre-Conceptual Schemas for Automatically Obtaining UML Diagrams", *Proc. of Revista Avances en Sistemas e Informática*, vol. 4, no. 2, pp. 117-124, 2007.

[8] E. Baksa-Varga, L. Kovács, "Linearization of the Extended Conceptual Graph Model with Dependency Grammar", ???

[9] M. G. Ilieva, "Graphical Notation for Natural Language and Knowledge Representation", *Proc. of 19th International Conference on Software Engineering and Knowledge Brooks Cole Publishing Engineering*, pp. 361-367, 2007.

[10] S. Mitra, "Object-Oriented+ Specification in VDM+", In: *Object-Oriented Specification Case Studies*, Prentice Hall International, 1994.

[11] J. F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks Cole Publishing, 2000.

[12] H. Liu, "Unpacking Meaning from Words: A Context-Centered Approach to Computational Lexicon Design", *Lecture Notes in Computer Science*, vol. 2680, pp. 218-232, 2003.

[13] Wasfy and Noor : Rule-based natural-language interface for virtual environments

[14] Myers, B. A., Pane, J. F. and Ko, A. (2004). Natural Programming Languages and Environments. *Communications of the ACM*, special issue on End-User Development, September, 47, 9, 47-52

[15] Mascardi V., Martelli M., Sterling L.: Logic-based Specification Languages for intelligent software agents (2003), "Theory and Practice of Logic Programming", volume 4,

[16] MOF model standard (http://www.omg.org/docs/html/06-01-01/Output/06-01-01.htm)

[17] Barbara H. Liskov and Stephen N. Zilles, Programming with Abstract Data Types, Computation Structures Group, Memo No 99, MIT, Project MAC, Cambridge Mass, 1974.

[18] Larman, C., Protected variation: the importance of being closed, Software, IEEE, Volume: 18, Issue: 3, On page(s): 89-91, Publication Date: May 2001.