UMLAUT: Synthesis of Natural Language from Constrained UML Models

Martin Ring[1], Jannis Stoppe[1,2,3], and Rolf Drechsler[1,2]

[1] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[2] Department of Mathematics and Computer Science, University of Bremen, Germany
[3] Institute for the Protection of Maritime Infrastructures, German Aerospace Center, Germany
martin.ring@dfki.de jannis.stoppe@dlr.de drechsle@informatik.uni-bremen.de

***Abstract*—Understanding formal specifications is a fundamental prerequisite in the design of any complex system. While it is feasible to machine check the conformance of an implementation to a specification, in practice many problems arise from the fact, that it is currently impossible for a machine to check the consistency of a specification in terms of the vision of a stakeholder, the requirements of concerned parties, or any laws which are not formalised. This paper presents an approach to generate natural language from formal specifications to aid in communicating complex technical matters to human beings.**

I. Introduction

The Unified Modelling Language (UML) has become the de-facto standard for modelling complex systems [4]. It provides not only a graphical language that enables designers to draw various kinds of diagrams to describe a given system but also provides a (mostly [1]) formal way to do so, removing ambiguities and opening the language to tools that process its designs further.

However, while UML has the reputation of an easily understandable modelling framework, it is far from self-explanatory and needs trained designers that are able to properly utilize and comprehend the according descriptions: while concepts such as classes and fields may be intuitively clear, e.g. the different types of diamond connectors and annotated quantities are usually not. With third parties that may be involved in projects (as client, partner or any other position), this training cannot be taken for granted, often resulting in specifications being written twice, once in UML and once in natural language, in order to be sure that any involved party may properly understand the issues around the project.

With the Object Constraint Language (OCL), matters get even more complex. OCL is used to annotate constraints into models: methods that may not be called unless certain conditions hold or must set certain values in a certain way are incorporated into UML designs using a (debatably) simple language. This turns UML into a language that not only describes a structure but also sets hard constraints concerning the described system's behaviour. There are various works that analyse the impact of OCL constraints and use it e.g. to verify that a design does not contain certain errors such as deadlocks. However, while enabling designers to add more information about the design and its behaviour to the model, OCL is also even less intuitively understandable than standard UML. While this does not doubt the usefulness of UML/OCL by itself, it is an issue to be addressed when sharing designs with people that are not trained and/or experienced enough to use it.

While there are attempts at linking a formal model to its natural language description [8], these attempts act mainly as a (more or less reliable) assistance that basically only highlight parts in the natural language document that may need to be addressed when a given part of the UML description has changed (and vice versa). This may be helpful for keeping the specification consistent but still means a considerable amount of work to keep both

specification means updated – essentially requiring designers to write the specification twice, once in natural language and once in UML.

However, while automatically synchronizing the two description means is still out of grasp, a one-way translation is quite possible. As UML/OCL is an unambiguous, formal description of a system it can be safely and automatically translated to natural language. The resulting text may serve as a means to communicate a given design to people who are not familiar with UML/OCL, reduce potential errors that may be introduced into the designs by inexperienced designers or simply serve as a way to "proofread" a design. While automatically generated natural language often suffers from e.g. repeating sentence structure or rather uninteresting flow, the intended target format (technical documentation) is already the prime example of these traits, maybe even turning this perceived disadvantage into a desired feature. This paper introduces the *UML Analysis and Understanding Tool* (UMLAUT), an implementation of a UML/OCL to natural language specification tool, starting with the preliminary topics of both UML/OCL and natural language in Section II, outlining the used methodology in detail in Section III and finally giving a case study of the approach in Section IV.

II. Preliminaries

This section describes the basics that are demanded to be able to generate proper english sentences out of UML and OCL constraints. For a comprehensive overview about NLP concepts and techniques the reader is referred to [3] and [2].

A. UML

The Unified Modelling Language (UML) is a graphical modelling language used to describe the architecture, inner structure as well as behavior of systems. The scope of this paper concentrates on class diagrams which are unarguably the most commonly used diagram type of the UML. Class diagrams are used to describe the semantic relations between classified domain objects. These may include abstract types which are not implementable, like the user of a system. They are however not at all usefull in describing behavioral properties of the model. For this kind of modelling there are other diagram types like state machine diagrams, activity diagrams or use case diagrams to name the most popular. However any property describable in one of these diagrams can be expressed as an OCL constraint on a class diagram. This is why we resorted to class diagrams + OCL for this work. A formal specification of the UML can be found in [6].

B. OCL

The Object Constraint Language (OCL) is used to describe non executable properties on e.g. a class diagram. The OCL can refer to other diagram types but is most commonly used to establish invariants of classes as well as pre and post conditions of operations. While the OCL is the de-facto standard for behavioral constraints on class diagrams, it is not without drawbacks. It's four valued logic as well as some syntactic idiosyncrasies make it not straight forward to understand for people not accustomed with it. [5] This makes the OCL an interesting target in the exploration of this paper as our aim is to generate natural language which does not require special knowledge about OCL syntax.

C. Natural Language Processing

Since we are not processing natural language but generating it, only a subset of this area is of relevance. In order to make our results compatible with existing tools and to increase the usefullness we do not generate strings of words but predictive sentence template (PST) trees.

III. Methodology

As outlined in Section I, the core idea of this work is to generate a natural language description of UML/OCL models. The self-evident origin for a workflow is thus an existing UML/OCL model. In this work, an existing formal model is thus assumed to be given.

While each element of this description could be mapped to predefined sentences with placeholders for e.g. variable names, this would result in issues for e.g. multiple pass approaches that rely on

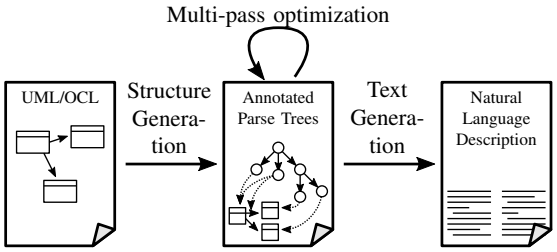

Fig. 1. UMLAUT's workflow

comparing sentences. Instead of merely generating sentence strings for a given structure, the workflow therefore relies on mapping the formal description elements to annotated parse trees as illustrated in Fig. 1. Nodes of the PST are annotated with references to their semantic origins, allowing to identify interconnections within the sentences without having to rely on error prone language analysis.

These structures are not only used to generate the readable sentences that compose the specification but may also serve as a foundation for other tools that work on the natural language representation or the underlying formal model. E.g. usually, any NLP tool needs to parse the text and interpret it on its own, but with pre-computed parse trees the text becomes much less ambiguous and may be processed more reliably.

The outcome of this workflow is a textual specification of the formal model that includes descriptions of both, all structural (UML) features and all constraints appended to the model using OCL. This description serves several purposes.

- It allows people that are not familiar with formal models to read the specification and understand the design.
- It allows designers to debug their designs by comparing an automatically generated description of the model to their own expectations.
- It allows NLP tools to rely on parse trees that are by definition unambiguous and correct.

Each of the steps (generation of parse trees, optimization of these structures and translation to text) is depending on underlying technologies and custom tools and techniques. These are outlined next.

A. UML foundation

Due to the fact that the UML is a visual language, parsing UML models is not always straightforward. In fact, it becomes virtually impossible when done with purely visual tools. Instead of building a custom solution, UMLAUT relies on an existing, versatile machine readable representation: the Eclipse Modeling Foundation (EMF) provides a representation of the UML, OCL and many UML profiles. All these modelling languages are representable in ECore, the interchange format of the EMF.

After a model has been parsed by faciliating the ECore interpreter provided by the EMF, we analyse the structure of the model. The ECore structure is then transformed by traversing it, passing around context information, which indicates the semantic setting of entities. The output of the transformation is an annotated parse tree, containing links to the original structure as well as dependency links between parts of the tree. From this output, we can obtain a pure parse tree as well as the original structure.

Since the links are combined in one structure we can be sure to generate consistent information and gain considerable confidence about the integrity of the result by comparing the original model to the reproduced one.

B. Optimization Passes

The generated parse trees are rough and monotonous, since every detail of the specification is transformed into a sentence. However, since we posess structured and annotated parse trees we are able to do optimisations after the generation phase. Especially repetition can be easily eliminated by matching sentence structures: If any set of sentences has the same subject and the same head in the verbial phrase, we can potentially transform it into a single sentence. There are however cases, where this kind of conflation is not desired, e.g. when the verb is *"must"* or *"should"* or when the complexity of the verbial phrase is high, since this would result in unpleasant sentences. Other optimisations are e.g. name elimination for cases where variables are introduces but not referenced (e.g. in ocl iterator expressions).

C. Natural Language Generation

An important aspect about the transformation into natural language is the interpretation of named entities. Our algorithm will not work when cryptical names or uncommon abbreviations are used within the model. Since UML models are supposed to be human readable we make appropriate namings a precondition for the application of our algorithm. However, there are a couple of aspects that can not be ignored:

a) Names dont contain whitespace: and thus designers resort to `camelCase`, `underscore_style` or `hyphen-style` identifiers. Thus we process the names and identify these patterns to break the names into a sequence of words.

b) Type names may represent phrases: Type names such as `MagneticCard` represent phrases, where `magnetic` acts a an ajective for the noun `card`. For types the defining nominal part is considered to be at the end of the identifier and all leading phrases are considered as constraining attributes. To detect the phrase boundaries we faciliate WordNet to obtain part of speech tags for the words in the identifier. This is not completely accurate and not applicable to very long names.

c) Value names may include type names: It is not uncommon, that names include an explicit reference to a type. There are two kinds of type references: the first is type repetition as in `Building {authorizedPerson: Person}` where the type of a property is repeated in the name; or `Card {cardOwner: Person}` where the enclosing type is repeated. Of the second kind are type abbreviations as in `card: MagneticCard`. Type repetitions usually occurr where names reflect constraining attributes of a type. In the previous examples, the name `authorizedPerson` refers a Person which is authorized for the building, whereas the `cardOwner` refers to a Person which is the owner of the card. A plain type abbreviation usually indicates that the function of the referred type is unambiguous and does not constrain the target type (the card of the person or the doors of the building). Type repetitions and type abbreviations may also occur in a mixed form. Therefore we explicitly search for type references within property names and then eliminate the repetition.

d) Operations may be active or passive acts: Unfortunately there is no broadly accepted guideline concering the placement of operations in UML classes. For operations with one parameter, there are always two possible interpretations. Consider the operations `enter` in `Building {enter(person: Person)}` and `Person {enter(building: Building)}`: For human readers it is obvious that both operations represents the situation where a person enters a building. However, for an algorithm there is no general rule and both operations might also represent a building entering a person. There is no way to disambiguate these cases without further semantic knowledge about the physical system represented by the names or statistical analysis. A simple approach to statistical analysis is to perform a web search with both generated phrases: `"person enters building"` yielded ~ 500 results on google, while `"building enters person"` yields zero results (in case of publication of this paper this it will yield ~ 1 result).

e) Stemmed lemmas: One important difference from the usual definition of PSTs is that our PST does not contain inflections. All leaves of the PST contain only the stem of their lemmas even if they indicate an inflected form. This allows us to separate the generation of data from grammatical details. The lemmas are derived from the names of the UML OCL entities. In a next step we can then generate sentences from the PSTs. For this, an algorithm was implemented which sanitises the leaves of the PST by applying the appropriate inflection. After that, the lemmas of the PST can be sequentialized into a string representing the natural language representation.

IV. Experimental Evaluation

In order to be able to evaluate our approach, we used several smaller UML/OCL benchmarks. The most complex exploration was a model of an access control system, as proposed in [7], illustrated in Fig. 2. The results of our algorithm can be inspected in Table I. The scientific evaluation of

these results is a work in progress at the time of this workshop and an issue which we would happily like to discuss with the audience. However, as a base for the discussion we outline our idea in the following.

Any information about the usefulness of our approach can only be obtained from humans, since our intention is to increase the comprehensibility for human readers. There are several interesting questions which arise

1) *Performance* – Does the generated natural language enable people (which are not previously accustomed with OCL) to understand a certain detail about the described system, which she or he would not if they read the OCL constraint. (Or understand the same detail in shorter time)?
2) *Accuracy* – Does the natural language description introduce systematic errors (wrong or missing information) in the understanding of the system, which plain OCL would not?
3) *Precision* – Are there people for which the natural langugage description is significantly less usefull than for other people or is there any other random error?

All three questions should be answered with a large user study. Important information to survey beforehand include primarily the prior experience with UML OCL or similar modelling languages. During the test, the probands should be divided into three groups. One should be given the original UML/OCL description for limited amount of time. The other two groups get to read the results of our algorithm as well as a manual description of the model. After the setting-in period the probands will be asked several questions of increasing dificulties about details of the model. (Is a described situation a violation of the specification, how many As belong to a B, how are A and B related, etc.). The results can be compared among the groups to answer all three questions above.

V. Conclusion

In this paper we introduced an algorithm that generates natural language sentences out of formal OCL descriptions. We have not yet scientifically evaluated our approach but are confident that it may be of some utility in certain situations. Especially to aid the understanding of a system design for non experts. So far the algorithm does handle a comparably small subset of UML/OCL. However, the subset is large enough to be used in the evaluation of the approach. If our approach turns out to be valid, future work must incorporate other common diagram types as well as more complex OCL constructs. Especially in the context of hardware systems, SysML diagrams (which are not easily mappable to UML diagrams like Class/Block diagrams) should be considered aswell.

Additionally, the quality of the generated language needs to be evaluated in detail. The core issue here is to find a reliable metric to grade a natural language specification with – or to conduct a larger scale study as to how the generated specification is perceived.

Acknowledgements

Financial support of subproject P02 “Heuristic, Statistical and Analytical Experimental Design” of the Collaborative Research Center SFB 1232 “Farbige Zustände” by the German Research Foundation (DFG), the Reinhart Koselleck project DR 287/23-1 (DFG), BMBF grant SELFIE, no. 01IW16001 is gratefully acknowledged.

References

[1] R. France, A. Evans, K. Lano, and B. Rumpe, “The uml as a formal modeling notation,” *Computer Standards & Interfaces*, vol. 19, no. 7, pp. 325–334, 1998.

[2] N. Indurkhya and F. J. Damerau, *Handbook of Natural Language Processing*, 2nd ed. Chapman & Hall/CRC, 2010.

[3] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Pearson Prentice Hall, 2008.

[4] A. Knapp and S. Merz, “Model checking and code generation for uml state machines and collaborations,” *Proc. 5th Wsh. Tools for System Design and Verification*, pp. 59–64, 2002.

[5] OMG, “Object Constraint Language, v2.4,” The Object Management Group, Tech. Rep., 2014.

[6] OMG, “Unified Modeling Language, v2.5,” The Object Management Group, Tech. Rep., 2015.

[7] N. Przigoda, J. Stoppe, J. Seiter, R. Wille, and R. Drechsler, “Verification-driven design across abstraction levels: A case study,” in *Euromicro Conference on Digital System Design (DSD)*. IEEE, 2015, pp. 375–382.

[8] M. Ring, J. Stoppe, C. Lüth, and R. Drechsler, “Change impact analysis for hardware designs from natural language to system level,” in *Forum on Specification and Design Languages (FDL)*, 09 2016, pp. 1–7.

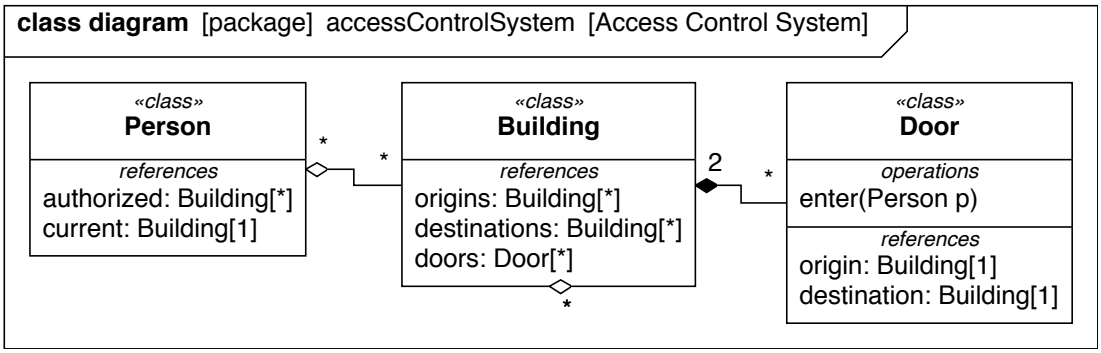

Fig. 2. Class Diagram of the Access Control System

```
context Person
        inv P5c: self.authorized->includes(self.current)
        inv P10c: self.authorized->forAll(b|self.authorized.destinations->includes(b))

context Building
        inv gate_defc: self.origins = self.doors.destination
        inv P7c: not (self.origins->includes(self))

context Door::enter(p: Person):
        pre enter_pre1c: p.authorized-> includes(self.destination)
        pre enter_pre2c: p.current = self.origin
        post enter_postc: p.current = self.destination
```

Fig. 3. OCL constraint to the access control system

TABLE I
RESULTING SENTENCES FOR THE ACCESS CONTROL SYSTEM BENCHMARK

1	the access control system cointains buildings, persons and doors.
2	every building has a set of origin buildings, a set of destination buildings and a set of doors.
3	the set of origins of a building must be equal to the set of destinations of all doors of the building.
4	the set of origins of a building must not include the building.
5	every person has a set of authorized buildings and a current building.
6	the set of authorized buildings of a person must include the current building of the person.
7	for every building B included in the set of authorized buildings of a person, the set of destinations of all authorized buildings of the person must include B.
8	every door has an origin building and a destination building.
9	every door can be entered by a person.
10	before a person P enters a door, the set of authorized buildings of P must include the destination of the door.
11	before a person P enters a door, the current building of P must be equal to the set of origins of the door.
12	after a person P entered a door, the current building of P must be equal to the set of destinations of the door.