Natural Language Generation from Class Diagrams

Håkan Burden
Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
burden@chalmers.se

Rogardt Heldal
Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
Gothenburg, Sweden
heldal@chalmers.se

ABSTRACT

A Platform-Independent Model (PIM) is supposed to capture the requirements specified in the Computational Independent Model (CIM). It can be hard to validate that this is the case since the stakeholders might lack the necessary training to access the information of the software models in the PIM. In contrast, a description of the PIM in natural language will enable all stakeholders to be included in the validation.

We have conducted a case study to investigate the possibilities to generate natural language text from Executable and Translatable UML. In our case study we have considered a static part of the PIM; the structure of the class diagram. The transformation was done in two steps. In the first step, the class diagram was transformed into an intermediate linguistic model using Grammatical Framework. In the second step, the linguistic model is transformed into natural language text. The PIM was enhanced in such a way that the generated texts can both paraphrase the original software models as well as include the underlying motivations behind the design decisions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement—*Documentation, Restructuring, reverse engineering and reengineering*; 1.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Language generation*; I.6.4 [**Simulation and Modeling**]: Model validation and analysis; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Reliability, Verification

Keywords

Model-Driven Architecture, Model Transformations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. INTRODUCTION

In Model-Driven Architecture (MDA; [15, 25]) software models are transformed into code in a series of transformations. The models have different purposes and level of abstraction towards the resulting implementation.

A Computational Independent Model (CIM) shows the environment of the software and its requirements in a way that can be understood by domain experts. The CIM is often referred to as the domain model and is specified using the vocabulary of the domain's practitioners and the stakeholders [17].

In the transformation from a CIM to a Platform Independent Model (PIM) the purpose of the models change and the focus is on the computational complexity that is needed to describe the behaviour and structure of the software.

The PIM is then transformed into a Platform Specific Model (PSM) which is a concrete solution to the problem as specified by the CIM. The PSM will include information about which programming language(s) to use and what hardware to deploy the executable code on.

One way of realising the model transformations in the MDA process is shown in Figure 1 which is adopted from [17]. In this process the transformation from CIM to PIM is done manually while the transformation from PIM to PSM is formalised by using marks and mappings. The marks reflect both unique properties of a certain PSM as well as domain-specific properties of the PIM, while the mappings describe a model to model transformation [15].

In MDA the PIM should be a bridge between the CIM and the PSM. Thus it is important that the PIM is clear and articulate [11, 33] to convey the intentions and motivations in the CIM as well as correctly describe the PSM [26]. Claims have been made that comprehensibility is more important than completeness if models are used for communication between stakeholders [18]. But if the stakeholders want to know if the PIM is complete with regards to the CIM, completeness is just as important.

1.1 Motivation

The developers of the PIM have to interpret the CIM to make their design decisions. Thus there are many ways for the PIM to represent a different solution to the problem compared to the solution given by the CIM: The CIM might be ambiguous or use vaguely defined concepts with the risk that it is misinterpreted; the CIM might be incomplete in the view of the developers of the PIM so they make additions to the PIM and finally, the CIM might be assessed as incorrect but the correction is made in the PIM and not in

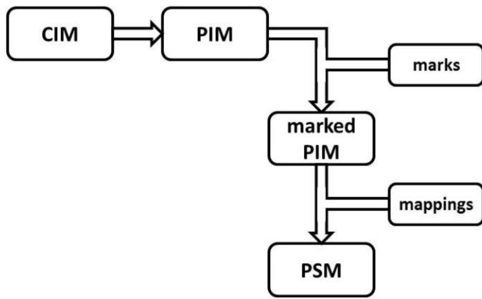

Figure 1: One realisation of the MDA process

the CIM. Over time the CIM and the PIM diverge due to the interaction of these inconsistencies.

The problem is not limited to the development phase. In order to adopt the CIM and the PIM to changing requirements, new developers have to be able to understand why the models are designed in the way they are and how they can be changed according to their underlying theory [20].

An example of the threat of failing to understand the underlying theory is given in [2]. From their experiences at British Airways they report on how important business rules are trivialised in the PIM as it is incapable of showing which business requirements are most important when all elements look the same in a class diagram. To demonstrate their point they use the notion of codesharing. Codesharing is when airlines in an alliance can sell seats on each others flights. For this to be possible a flight has to be able to have more than one flight code. In a class diagram this business requirement worth millions of pounds is obscured as a simple multiplicity on an association between two classes, see Figure 2.

So the transformation from CIM to PIM poses two questions: How do we know that the PIM captures the requirements of the CIM, and nothing else? And how can we make sure that future developers of the PIM understand the intentions and motivations behind the design decisions [20]? The evaluation of the correctness of the PIM's behaviour and structure can be done by testing and model reviewing.

Both testing and accessing the information of the PIM requires an understanding of object-oriented design, knowledge of the used models and experience of using tools for software modelling [2]. Textual descriptions, on the other hand, are suitable for stakeholders without the necessary expertise in software models [9]; natural language can be understood by anyone, allowing all stakeholders to contribute to the validation of the PIM.

1.2 Aim

Our long-term aim is to reverse engineer the marked PIM into a CIM, investigating how much of the original CIM that can be generated from the marked PIM. As our first step towards a complete system we have chosen the structure of the class diagrams. The aim of the generated text is not only to paraphrase the class diagram but also to include the underlying motivations and design decisions that form the theory behind the model.

By using an MDA approach for generating natural language text we enable the textual description of the PIM, the PIM itself and deployed PSMs to be synchronised with each other. The texts can be used by stakeholders that are unfamiliar with software models to validate the structure and behaviour of the models, enabling a process that leads to software meeting the requirements and expectations of all stakeholders.

1.3 Contribution

We have generated textual descriptions of the structure of the class diagram that not only paraphrase the diagrams but also include the underlying motivations and design decisions. The mappings from marked PIM to natural language PSMs are generic and can be applied to any marked PIM. Indeed, since the marks are used to enhance the performance of the mappings the transforming an unmarked PIM will still generate a linguistic model. Though the text generated from such a linguistic model might have minor grammatical errors.

The vocabulary of the PIM is reused as lexicon for the generated linguistic model so that we can generate text for any domain independent of how technical or unpredictable the vocabulary may be.

In MDA terms the generation of natural language was solved by first transforming the xtUML models into an intermediate linguistic model, a grammar. In a second transformation the grammar was used to generate the desired view of the class diagrams as natural language text.

1.4 Overview

In the next section we present the background knowledge for our case study in terms of natural language generation, the Grammatical Framework and Executable and Translatable UML. In section 3 we describe our case study of transforming the PIM into a CIM. The results are given in section 4, followed by a discussion in section 5. Our case study is related to previous work in section 6 and a summary with drafts for future work concludes our contribution.

2. BACKGROUND

In our case study we have used the MDA perspective on models for Natural Language Generation [29]. This was achieved by first transforming the marked PIM into a linguistic model defined by the Grammatical Framework [28]. The linguistic model was then used to generate the final textual description of the PIM. We used Executable and Translatable UML to model the class diagram and the model to model transformation.

2.1 Executable and Translatable UML

The Executable and Translatable Unified Modeling Language (xtUML; [14, 27, 34]) evolved from merging the Shlaer-Mellor method [30] with the Unified Modeling Language (UML, [23]).

There are three kinds of diagrams used in xtUML (component diagrams, class diagrams and statemachines) as well as a textual Action language. The Action language is used to define the semantics of the graphical diagrams. This study only concerns the class diagrams.

2.1.1 xtUML Class Diagrams

In Figure 2 we have an example of an xtUML class diagram. The xtUML classes and associations are more restricted than in UML. We will only mention those differences that are interesting for our case study.

In UML the associations between classes can be given a descriptive association name while in xtUML the association

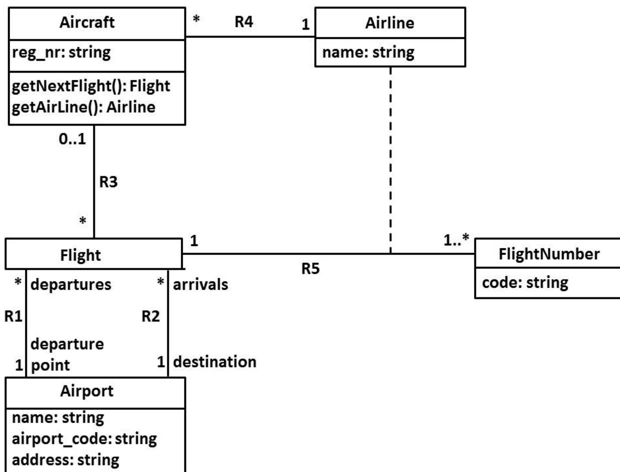

Figure 2: An xtUML class diagram

names are automatically given names on the form `RN` where `N` is a unique natural number. I.e. `Flight` is associated to `FlightNumber` over the association `R5`.

In xtUML there are no special associations for the UML aggregate and composition associations. Both aggregation and composition express a parts-of relation with the difference that in aggregation, the parts can exist without a 'whole' while in composition the parts cannot exist without the 'whole'. Following the definition given by the OMG [23] aggregation is modelled by using the multiplicity 0..1 and composition by using the multiplicity 1.

Speaking of multiplicities, in xtUML there are only four possible combinations of multiplicities; 0..1, 1, * and 1..*.

2.1.2 Model Transformation

The PIM to PSM transformation is handled by model compilers. A model compiler takes a marked PIM and a set of mappings that specify how the different elements of the marked PIM are to be translated into the PSM [15, 17]. Since the PSM is generated from the marked PIM, it is possible for the running code and the software models to always be in synchronization with each other since all updates and changes to the system are done at the PIM-level, never by touching the PSM. The model compiler allows the same PIM to be transformed into different PSMs [1] without a loss in efficiency compared to handwritten code [31].

2.2 Natural Language Generation

When compiling a marked PIM into a PSM it is important to include all the information of the marked PIM into the transformation. For Natural Language Generation (NLG) this is not the case [29]. The content, its layout and the internal order of the generated text is dependent on who the reader is, the purpose of the text and by which means it is displayed. In this sense the texts can be seen as platform-specific.

Traditionally NLG is broken down into a three-stage pipeline; text planning, sentence planning and linguistic realisation [29]. From an MDA perspective NLG can be viewed as two transformations. The first transformation takes the software model and reshapes it to an intermediate linguistic model by performing text and sentence planning. The second transformation is equivalent to the linguistic realisation as the linguistic model is transformed into natural language text. We will use our class diagram in Figure 2 to exemplify the purpose of the three stages.

2.2.1 Text Planning

Text planning is to decide on what information in the original model to communicate to the readers. When the selection has been done the underlying structure of the content is determined. In our case we first describe the classes with attributes and operations, then the associations between the classes with multiplicities.

2.2.2 Sentence Planning

When the overall structure of the text is determined the attention is turned towards the individual sentences. This is also the time for choosing the words that are going to be used for the different concepts, e.g. an aircraft can both *depart* or *leave* an airport. The original software model has now been transformed into a linguistic model.

2.2.3 Linguistic Realisation

In the last stage the linguistic model is used to generate text with the right syntax and word forms. The linguistic model should ensure that the nouns get the right plural forms and that we get ***a** flight* but ***an** aircraft*. Through the linguistic realisation the intermediate model has been transformed into a natural language text.

2.3 Grammatical Framework

For defining the linguistic model we use Grammatical Framework (GF, [28]). In GF the grammars are separated into an abstract and a concrete syntax. To understand how we have used GF and the resource grammars we give an example that generates the sentence *An Aircraft has many Flights*. The grammar is found in Figure 3. It is not necessary to understand the details of the grammar, it is included as a small example of the kind of output that is generated from our model to model transformation.

2.3.1 Abstract Syntax

The abstract syntax is defined by two finite sets, categories (`cat`) and functions (`fun`). The categories are used as building blocks and define the arguments and return values of the functions

From the class diagram in Figure 2 we have that both `Aircraft` and `Flight` are class names. We want to use this information in our grammar, defining a function for both `Aircraft` and `Flight`, see Figure 3. From a linguistic point of view they define the lexical items that make up our lexicon. Lexical items can be used to define more complex functions, like `OneToMany` that returns a `Text` describing the association between two `ClassNames`. By defining our categories (the content of the text) and the functions (the ordering of the content) we have completed the text planning stage of the NLG process.

2.3.2 Abstract Trees

Abstract syntax trees are formed by using the functions as syntactic constructors according to their arguments. While the abstract syntax shows the text planning for a possibly

Abstract syntax:

```
cat Text, ClassName ;

fun Aircraft : ClassName ;
    Flight : ClassName ;
    OneToMany : ClassName × ClassName → Text ;
```

Concrete syntax:

```
lincat Text = RGL.Text ;
       ClassName = CN ;

lin Aircraft =  mkCN (mkN "Aircraft" "Aircraft") ;
    Flight = mkCN (mkN "Flight") ;
    OneToMany aircraft flight =
      mkText (mkCl (mkNP (mkDet a_Quant) aircraft)
             (mkV2 have_V)
             (mkNP (mkDet many_Quant) flight)) ;
```

Figure 3: An example of an automatically generated GF grammar

infinite set of texts the abstract tree represents the structure of exactly one text. According to our example grammar the sentence *An Aircraft has many Flights* will have `OneToMany(Aircraft, Flight)` as its abstract tree.

2.3.3 Concrete Syntax

A concrete syntax assigns a linearisation category (`lincat`) to every abstract category and a linearisation rule (`lin`) to every abstract function. The linearisation categories define how the concepts of the PIM are mapped to the pre-defined categories of GF. From an NLG perspective the linearisation rules supply the sentence planning. The concrete syntax is implemented by using the GF Resource Grammar Library.

2.3.4 Resource Grammar Library

In the Resource Grammar Library (RGL) a common abstract syntax has sixteen different implementations in form of concrete syntaxes. Among the covered languages are English, Finnish, Russian and Urdu. The resource grammars come with an interface which hides the complexity of each concrete language behind a common abstract interface.

The RGL interface supplies a grammar writer with a number of functions for defining a concrete syntax. In Figure 3 `mkText`, `mkCl` and `a_Quant` are examples of such functions. Exactly how these functions are implemented is defined by the concrete resource grammar for each language. Just as for a programming language we only need to understand the interface of the library to get the desired results, we do not need to understand the inner workings of the library itself.

2.3.5 Linearisation

In GF the linearisation of an abstract tree, `t`, by a concrete syntax, `C`, can be written as $\mathtt{t}^{\mathtt{C}}$ and formulated as follows

$$(\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n))^{\mathtt{C}} = \mathtt{f}^{\mathtt{C}}(\mathtt{t}_1^{\mathtt{C}}, \ldots, \mathtt{t}_n^{\mathtt{C}})$$

where $\mathtt{f}^{\mathtt{C}}$ is a concrete linearisation of a function `f` [13].

The linearisation of `OneToMany(Aircraft, Flight)` using the concrete English grammar `ENG` described in Figure 3 is then unwrapped as follows

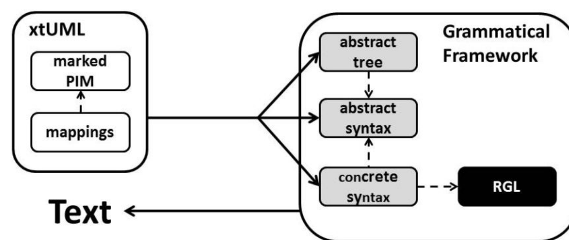

Figure 4: From marked PIM to text

$$\begin{aligned}
&(\mathtt{OneToMany}(\mathtt{Aircraft}, \mathtt{Flight}))^{\mathtt{ENG}} \\
&\quad = \mathtt{OneToMany}^{\mathtt{ENG}}(\mathtt{Aircraft}^{\mathtt{ENG}}, \mathtt{Flight}^{\mathtt{ENG}}) \\
&\quad = \mathtt{mkText}(\mathtt{mkCl}(\mathtt{mkNP}(\mathtt{mkDet\ a_Quant})\ \mathtt{Aircraft}^{\mathtt{ENG}}) \\
&\quad\ (\mathtt{mkV2\ have_V}) \\
&\quad\ (\mathtt{mkNP}(\mathtt{mkDet\ many_Quant})\ \mathtt{Flight}^{\mathtt{ENG}})) \\
&\quad = \mathtt{mkText}(\mathtt{mkCl}(\mathtt{mkNP}(\mathtt{mkDet\ a_Quant}) \\
&\quad\ (\mathtt{mkCN}(\mathtt{mkN}\ \text{"}\mathtt{Aircraft}\text{" "}\mathtt{Aircraft}\text{"})))) \\
&\quad\ (\mathtt{mkV2\ have_V}) \\
&\quad\ (\mathtt{mkNP}(\mathtt{mkDet\ many_Quant})(\mathtt{mkCN}(\mathtt{mkN}\ \text{"}\mathtt{Flight}\text{"})))) \\
&\quad = \textit{An Aircraft has many Flights}
\end{aligned}$$

Linearisation is an built-in functionality of GF and equivalent to the linguistic realisation of NLG.

3. NATURAL LANGUAGE GENERATION FROM CLASS DIAGRAMS

To investigate the possibilities for natural language generation from software models we have conducted a case study using xtUML to model the PIM and perform the model-to-model transformations. The reason for choosing xtUML is that the model compiler enables a convenient way of transforming the PIM to different PSMs. We used BridgePoint [3, 14] as our xtUML tool.

3.1 Case Description

The original case was a hotel reservation system. To avoid getting into domain details and explaining the different components and subsystems we reuse the example given in [2] with a small extension; we have added classes for the concepts `Aircraft`, `Airport` and `Airline`. The result is a class diagram that highlights the problems we want to solve and what we can achieve in forms of NLG. The class diagram can be found in Figure 2. The intention of the diagram is not a complete description of the problem domain.

Our PIM includes a note for the association `R5`, *A Flight can have more than one Flight number since code sharing is a multimillion-pound business, affecting an alliance of airlines.* There are also notes on the associations so that they carry meaningful association names instead of xtUML's generic ones. `R1` and `R2` are annotated with *has*, `R3` is annotated with *is booked for*, `R4` is annotated with *belongs to* which is to be read from left-to-right only and `R5` has the note *is identified by* which also is to be read from left-to-right.

An overview of our system is found in Figure 4. The shaded modules are generated in the model-to-model trans-

formation. The Resource Grammar Library (RGL) supplies the necessary details to realise the concrete syntax. The dotted lines within the systems give the dependencies between the modules while the solid lines show the transformations between the systems. The transformation between xtUML and Grammatical Framework is defined as mappings in BridgePoint while the transformation from Grammatical Framework to text is automatically handled by GF through linearisation.

The input to the first transformation in Figure 4 is a marked PIM and a set of mappings. The marks are described next and then the mappings.

3.1.1 Marking the PIM

Since we are aiming for a linguistic model and not source code we use marks for irregular word forms, where the marks play a similiar role as stereotypes in UML. In our example we use a mark on the class `Aircraft` so that the noun *Aircraft* has the same form in both singular and plural. Just as for UML the xtUML metamodel can be extended for different profiles. Our extension results in a natural language profile for xtUML. The general mapping is otherwise to use the regular form for English nouns, i.e. a plural s. The mappings are generic and can be used for any marked PIM.

3.1.2 Mappings

We use the following pseudo-algorithm to decide what the linguistic model should contain and in what order. These mappings are generic and can be used for any marked PIM. The mappings only consider certain aspects of the class diagrams of the PIM and if it contains other diagrams or action language this information is just omitted.

```
generate lexicon for class diagram;

for each class in class diagram
   if class has attributes
      generate sentence for class attributes ;
   if class has operations
      generate sentence for class operations ;

for each association in class diagram
   if association has association name
      generate sentences for association ;
   if association has association class
      generate sentence for association class ;
   if association has motivation
      generate sentence for motivation ;
```

The algorithm is implemented by using the xtUML model compiler.

3.2 xtUML to GF

3.2.1 Lexicon generation

Before we generate the different sentences of our text we need a vocabulary. The content of the vocabulary, or lexicon in linguistic terms, is taken from the names of the elements of the class diagram and the marking model. The lexicon therefore defines which concepts that will be included in the final text (flights, names, codes etc.) and for which reason (as class names, attributes and so on). Here is the automatically generated abstract syntax of the lexicon, in a dense representation to save space.

```
cat ClassName, Association, Attribute,
    Multiplicity, Operation, Motivation ;

fun Flight, FlightNumber, Aircraft, Airline,
        Airport : ClassName ;
    R1, R2, R3, R4, R5 : Association ;
    Name, Code, RegNr, Address,
        AirportCode : Attribute ;
    One, ZeroOne, ZeroMore,
        OneMore : Multiplicity ;
    GetNextFlight, GetAirline : Operation ;
    R5Motivation : Motivation ;
```

3.2.2 Classes

To list the attributes of a class we generate a unique abstract function for each class with one `Attribute` argument for each class's attribute in the PIM. The function corresponding to the class `Airport` has the following abstract syntax

```
AirportAttributes : ClassName × Attribute ×
     Attribute × Attribute → Text ;
```

At the same time we generate an abstract syntax tree for the function given the class it paraphrases

```
AirportAttributes(Airport, Name,
                  AirportCode, Address)
```

The same procedure as for attributes is repeated for listing the operations of the classes.

3.2.3 Associations

We generate one function for all associations

```
Association : Association × Multiplicity ×
    ClassName × Multiplicity × ClassName →
    Text ;
```

This function is a generalisation of the `OneToMany` found in Figure 3. For the association between `Flight` and `Flight Number` we get the following tree

```
Association(R5, One, Flight, OneMore, FlightName)
```

To generate a text for an association class we use one function that takes three class names as arguments

```
AssociationClass : ClassName × ClassName ×
    ClassName → Text ;
```

For each association with an association class we then generate an abstract syntax tree. For association R5 in our example diagram we get the following tree

```
AssociationClass(Flight, FlightName, Airline)
```

Each motivation is introduced into the grammars by a unique function and abstract tree

```
R5Text : Motivation → Text ;
R5Text(R5Motivation)
```

3.2.4 Combining texts

We now have a set of unconnected abstract trees. To combine the trees into one text we introduce the function

```
Combine : Text × Text → Text ;
```

If we append the generated abstract trees above, we get the following abstract tree

```
Combine(
   AirportAttributes(Airport, Name,
                     AirportCode, Address),
   Combine(
      Association(R5, One, Flight,
                  OneMore, FlightNumber),
      Combine(
         AssociationClass(Flight, FlightName,
                          Airline)
         R5Text(R5Motivation))))
```

We have now automatically transformed the class diagram into an abstract and a concrete syntax as well as an abstract syntax tree. Together these three represent a linguistic model of the text that we want to generate.

3.3 GF to Text

The generated abstract syntax tree for the document is linearised by the GF lineariser. The linearisation of the tree completes the transformation of our xtUML class diagram into natural language text.

4. RESULTS

To show the results from our NLG process we give a small text that is generated from the examples used in the previous section.

An Airport has a name, an airport code and an address. An Aircraft can get next Flight and get Airline. A Flight is identified by one or more Flight Numbers. The relationship between a Flight and a Flight Number is specified by an Airline. A Flight can have more than one Flight number since code sharing is a multimillion-pound business, affecting an alliance of airlines.

The generated text can now be used by the stakeholders to validate that the class diagram has the right structure and that the underlying theory is represented. The generation of textual descriptions from the class diagram enables close communication with stakeholders, giving them constant feedback which is a crucial point according to [9].

The grammars were automatically transformed from the class diagram, all we needed to do was to mark the PIM and give the mappings between the marked PIM and the grammar. To generate text from another class diagram we need new marks for the irregular nouns. We can then reuse the mappings defined in our example to generate natural language text from any marked PIM.

Since the role of the marks is to enhance the quality of the transformation defined by the mappings it is not necessary to start with a marked PIM. The results of applying the mappings to an unmarked PIM is that we get a grammar treating all class names as regular nouns. This might lead to some odd phrasings, such as *many Aircrafts*. The division of labour between marks and mappings means that a developer with a reasonable knowledge of English can mark the PIM with the necessary irregularities while an expert on the target langauge and the used grammar formalism can define the mappings once and for all.

A further result is that we managed to combine two different systems that are successful within their respective domains. Executable and Translatable UML (xtUML) has previously been proven to allow the PIM and the PSM to be consistent with each other as well as enabling reuse [1, 31]. GF is currently used in collaboration with industry for multilingual translation in the MOLTO-project [19] and has previously been used for multi-modal dialogue systems [4, 35] and in collaboration with the car industry [12].

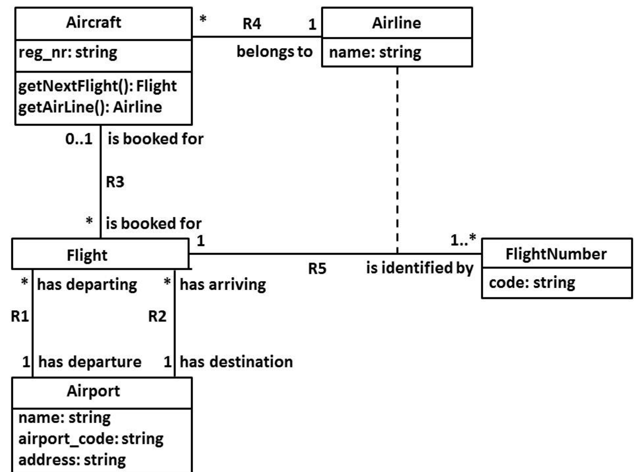

Figure 5: An xtUML class-diagram

5. DISCUSSION

In our Motivation we stressed that even a well-formed model is difficult to understand, thus the need for textual paraphrasing of its content and motivations. On the other hand, paraphrasing the model will not make up for a lack of detail in the model, those details are needed to make the text informative. It is therefore important that the models use meaningful names for classes, attributes and associations etc. so that it is possible to generate a precise vocabulary and meaningful descriptions of why classes are associated with each other.

In UML we can use verbs or verb phrases for the association names and nouns for the role names [16]. The role names can thus be seen as outsourced attributes. The problem is how to incorporate the information given by the class name and the role name together with the association name. For the class diagram in Figure 2, we state that *An Airport has one or more arrivals.* But what is an arrival? A clarification can be done in many ways, one is by adding subordinate clauses that define an arrival, *where an arrival is a Flight.* In xtUML the issue is solved differently.

Associations are given default names in xtUML, names that have no semantic meaning to a human reader. To understand what the association represents one has to understand the Action language that defines the association. The lack of a verb phrase for the association opens up a new way of looking at role names; [33] advocate that the role names should be used as underspecified verb phrases that are missing their complement. By using this definition of role names on our class diagram we get a new diagram adopted to xtUML, see Figure 5. The benefit is that we do not need to mark the associations to give them meaningful names and we can use the roles of the classes at the same time. From this diagram we could generate the sentence *An Airport has one or more arriving Flights.*

6. RELATED WORK

In a systematic literature review from 2009 there is only one work that reports on generating natural language text from class diagrams. From our own searches we have not found any MDA approach that cites the review. However there are other contributions that have used the same techniques as we have, but in other settings.

A systematic literature review on text generation from software engineering models is reported in [21]. Of the 24 contributions only one concerned the generation of natural language text from UML diagrams, [16]. The motivation for conducting the literature review was that even if models are precise, expressive and widely understood by the development team natural language has its benefits. Natural language enables the participation of all stakeholders in the validation of the requirements and makes it clear how far the implementation of the requirements have come. [21] state that none of the contributions address the issue of keeping the generated documents synchronized with the PIM.

Our examples of generated text are inspired by the work done by [16]. They generate natural language descriptions of UML class diagrams using WordNet [8] for obtaining the necessary linguistic knowledge. WordNet is a wide-coverage resource which makes it useful for general applications but can limit the use for domain-specific tasks. We use a domain-specific grammar that is tailored for just our needs. Whatever the domain our approach has lexical coverage while WordNet will lack lexcial knowledge about more technical areas. When it comes to results there texts are descriptions of the class diagram while ours also include the underlying motivations for the structure.

In [7] the Semantics of Business Vocabulary and Business Rules (SBVR, [24]) is used as an intermediate representation for transforming UML and OCL into constrained natural language. This means that SBVR maps to a limited set of possible sentence structures while GF allows a free sentence planning.

[5] have developed a system that transforms class diagrams into natural language texts. Their system differs from ours in that it marks all model elements with the corresponding linguistic realisation. While our system relies on the linguistic model to perform the linguistic realisation, their system maps the marks straight into pre-defined sentences with slots for the linguistic realisation of the model elements.

Grammatical Framework has been used before to generate requirements specifications [6, 10] in the Object Constraint Language (OCL; [22, 36]). GF is used to translate expressions in OCL to English text with LaTeX-formatting. The translation is done by implementing an abstract grammar for the UML model of OCL, a concrete grammar for OCL expressions and a concrete grammar for English. The text to text translation is then done by obtaining an abstract tree through parsing the OCL-expression, then linearizing the tree in English. Since we do not have a grammar for our graphical models we instead use the metamodel of xtUML to generate the necessary linearisation grammars.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

From our generated text it is possible to see if the motivations and intentions of the CIM are captured by the PIM. The texts also paraphrases the structure of the class diagram, enabling stakeholders with various backgrounds to participate in the validation of the PIM. In the process we have transformed the class diagram into an intermediate linguistic model which ensures that the generated texts are grammatically correct.

7.2 Future Work

From our case study we have identified two lines of future work that we find interesting. The first line is to generate other views of the PIM, the second line is to make more use of the Grammatical Framework.

So far we have looked at the static structure of the class diagram. Another aspect worth looking in to is the dynamic behaviour of the software. This can be done by transforming the Action language code into textual comments, adopting the results from [32] to xtUML and MDA. This will then be combined with natural language descriptions of the statemachines since they play a key role in the behaviour of objects.

There are several ways to make more use of GF. [6] enrich their generated texts with LaTeX, something that could be used to highlight the motivations or for supplying tags for colour and fonts to the texts. We also want to make more use of GF's capacity for several concrete languages to share the same abstract syntax. Being able to generate a variety of languages from internal system specifications would mean that the models can be accessed and evaluated by those stakeholders that are not confident in using English. One of the new languages could be a formal language for writing requirements and then GF could be used to both generate natural language descriptions, formal requirements and translate between the two.

Both lines of work will in the end require a more rigorous evaluation, both to obtain the desired format and content of the texts but also to see in which extent they can replace the original CIM.

Acknowledgments

The authors want to thank the Graduate School of Language Technology for partially funding our work. Toni Siljamäki at Ericsson AB and Leon Moonen at Simula Research Laboratory gave comments and tips on issues concerning MDA while Peter Ljunglöf and Aarne Ranta at Computer Science and Engineering gave advice on issues concerning Natural Language Generation and Grammatical Framework.

8. REFERENCES

[1] S. Andersson and T. Siljamäki. Proof of concept - reuse of PIM, experience report. In *SPLST'09 & NW-MODE'09: Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, August 2009.

[2] J. Arlow, W. Emmerich, and J. Quinn. Literate Modelling - Capturing Business Knowledge with the UML. In *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*, pages 189–199, London, UK, 1999. Springer-Verlag.

[3] BridgePoint. http://www.mentor.com/. Accessed 8th March 2011.

[4] B. Bringert, R. Cooper, P. Ljunglöf, and A. Ranta. Multimodal dialogue system grammars. In *Proceedings*

of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, pages 53–60, June 2005.

[5] P. Brosch and A. Randak. Position paper: m2n-a tool for translating models to natural language descriptions. *Electronic Communications of the EASST*, Software Modeling in Education at MODELS 2010(34), 2010.

[6] D. A. Burke and K. Johannisson. Translating formal software specifications to natural language. In P. Blache, E. P. Stabler, J. Busquets, and R. Moot, editors, *LACL*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2005.

[7] J. Cabot, R. Pau, and R. Raventós. From uml/ocl to sbvr specifications: A challenging transformation. *Inf. Syst.*, 35(4):417–440, 2010.

[8] C. Fellbaum and G. A. Miller. *WordNet: An electronic lexical database.* MIT Press, Cambridge, MA, 1998.

[9] D. Firesmith. Modern requirements specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[10] R. Hähnle, K. Johannisson, and A. Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2002.

[11] C. F. J. Lange, B. D. Bois, M. R. V. Chaudron, and S. Demeyer. An experimental investigation of UML modeling conventions. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[12] S. Larsson and J. Villing. The dico project: A multimodal menu-based in-vehicle dialogue system. In *Proceedings of the 7th International Workshop on Computational Semantics (IWCS-7), Tilburg, The Netherlands.* IWCS, 2007.

[13] P. Ljunglöf. Editing syntax trees on the surface. In *Nodalida'11: 18th Nordic Conference of Computational Linguistics*, volume 11, Riga, Latvia, 2011. NEALT Proceedings Series.

[14] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] S. J. Mellor, S. Kendall, A. Uhl, and D. Weise. *MDA Distilled.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[16] F. Meziane, N. Athanasakis, and S. Ananiadou. Generating Natural Language Specifications from UML Class Diagrams. *Requir. Eng.*, 13(1):1–18, 2008.

[17] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[18] P. Mohagheghi and J. Aagedal. Evaluating quality in model-driven engineering. In *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Molto - Multilingual On-line Translation. http://www.molto-project.eu/. Accessed 1st July 2011.

[20] P. Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253 – 261, 1985.

[21] J. Nicolás and J. A. T. Álvarez. On the generation of requirements specifications from software engineering models: A systematic literature review. *Information & Software Technology*, 51(9):1291–1307, 2009.

[22] OMG. Object Constraint Language Version 2.2. http://www.omg.org/spec/OCL/2.2/. Accessed 13th September 2010.

[23] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. http://www.omg.org/spec/UML/2.3/. Accessed 11th September 2010.

[24] OMG. *Semantics of Business Vocabulary and Rules (SBVR) Version 1.0*, ormal/08-01-02 edition, January 2008.

[25] OMG. MDA. http://www.omg.org/mda/, Accessed January 2011.

[26] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[27] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[28] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars.* CSLI Publications, Stanford, 2011.

[29] E. Reiter and R. Dale. Building applied natural language generation systems. *Nat. Lang. Eng.*, 3:57–87, March 1997.

[30] S. Shlaer and S. J. Mellor. *Object lifecycles: modeling the world in states.* Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[31] T. Siljamäki and S. Andersson. Performance benchmarking of real time critical function using BridgePoint xtUML. NW-MoDE'08: Nordic Workshop on Model Driven Engineering. Reykjavik, Iceland, August 2008.

[32] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[33] L. Starr. How to build articulate UML class models. http://knol.google.com/k/leon-starr/how-to-build-articulate-uml-class-models/2hnjef6cmm97l/4. Accessed 24th November 2009.

[34] L. Starr. *Executable UML: How to Build Class Models.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[35] The TALK Project. http://www.talk-project.org/. Accessed 1st July 2011.

[36] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.