Thesis for the Degree of Licentiate of Philosophy

Three Studies on Model Transformations – Parsing, Generation and Ease of Use

Håkan Burden

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden

Gothenburg, 2012

Three Studies on Model Transformations – Parsing, Generation and Ease of Use
© Håkan Burden, 2012

Technical Report no. 92L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Language Technology

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31–772 1000

Printed at Chalmers University of Technology, Gothenburg, 2012

ABSTRACT

Transformations play an important part in both software development and the automatic processing of natural languages. We present three publications rooted in the multi-disciplinary research of Language Technology and Software Engineering and relate their contribution to the literature on syntactical transformations.

Parsing Linear Context-Free Rewriting Systems

The first publication describes four different parsing algorithms for the mildly context-sensitive grammar formalism Linear Context-Free Rewriting Systems. The algorithms automatically transform a text into a chart. As a result the parse chart contains the (possibly partial) analysis of the text according to a grammar with a lower level of abstraction than the original text. The uni-directional and endogenous transformations are described within the framework of parsing as deduction.

Natural Language Generation from Class Diagrams

Using the framework of Model-Driven Architecture we generate natural language from class diagrams. The transformation is done in two steps. In the first step we transform the class diagram, defined by Executable and Translatable UML, to grammars specified by the Grammatical Framework. The grammars are then used to generate the desired text. Overall, the transformation is uni-directional, automatic and an example of a reverse engineering translation.

Executable and Translatable UML – How Difficult Can it Be?

Within Model-Driven Architecture there has been substantial research on the transformation from Platform-Independent Models (PIM) into Platform-Specifc Models, less so on the transformation from Computationally Independent Models (CIM) into PIMs. This publication reflects on the outcomes of letting novice software developers transform CIMs specified by UML into PIMs defined in Executable and Translatable UML.

Conclusion

The three publications show how model transformations can be used within both Language Technology and Software Engineering to tackle the challenges of natural language processing and software development.

Acknowledgements

First of all I want to thank my supervisors; Aarne Ranta, Rogardt Heldal and Peter Ljunglöf. I am indebted to their inspiration and patience. I also want to acknowledge the various members of my PhD committee at Computer Science and Engineering; Bengt Nordström, Robin Cooper, David Sands, Jan Jonsson, Koen Claessen and Jörgen Hansson.

There are two research environments that I particularly want to mention. The first is the Swedish National Graduate School of Language Technology, GSLT, where Robin Cooper and Joakim Nivre have played decisive parts in my graduate studies. Through GSLT I have had the benefit of attending numerous seminars and courses as well as enjoying stimulating discussions with the involved researchers. GSLT has also funded my position as graduate student. The second research environment is the Center for Language Technology at the University of Gothenburg, CLT. In my progression as researcher CLT has served the same role as GSLT but at a local, and more frequent, level. CLT has also funded the travelling involved in presenting one of the publications included in this thesis.

There are far to many people at Computer Science and Engineering, CSE, to mention you all but I'm grateful for all the talks we've had in the corridors and over the coffee machine. A special thank you to the technical and administrative staff at GSLT, CLT and CSE who have made my scientific life so much easier. I have also had some outstanding room mates over the years; Björn Bringert, Harald Hammarström, Krasimir Angelov, Ramona Enache and Niklas Mellegård. Thanks! It's been a pleasure sharing office space with you.

There are some researchers and professionals in the outside world that deserve to be mentioned; Tom Adawi, Toni Siljamäki, Martin Lundquist, Leon Starr, Stephen Mellor, Staffan Kjellberg, Dag Sjøberg and all anonymous reviewers. you've all helped me to become a better researcher and scholar.

On the private side I want to thank Ellen Blåberg, Malva Bukowinska Burden, Vega Blåberg, Tora Burden Blåberg and Björn Blåberg for keeping it real. Your encouragement and support has meant a lot. The same goes to Ingrid Burden, Tony Burden, Lars Josefsson och Christel Blåberg. And a big thanks to my numerous friends and relatives who keep asking me what I do for a living.

Tack Malva, Vega, Tora och Björn för att ni finns, tack för alla fina presenter som gör kontoret så vackert och arbetsdagen så mycket roligare. Utan er hade det inte varit möjligt.

Contents

Introduction **1**
1 Introduction . . . 1
1.1 Language Technology . . . 1
1.2 Software Engineering . . . 2
1.3 Transformations . . . 2
1.4 Thesis Overview . . . 2
2 Transformations and Translations . . . 3
3 Parsing Linear Context-Free Rewriting Systems . . . 5
3.1 Introduction . . . 5
3.2 Transformations . . . 6
3.3 Contribution . . . 9
4 Natural Language Generation from Class Diagrams . . . 9
4.1 Introduction . . . 9
4.2 Transformations . . . 10
4.3 Contribution . . . 12
5 Executable and Translatable UML – How Difficult Can it Be? . . . 13
5.1 Introduction . . . 13
5.2 Transformations . . . 13
5.3 Contribution . . . 14
6 Future work . . . 14
7 Conclusion . . . 14

Paper 1: Parsing Linear Context-Free Rewriting Systems **21**
1 Introductory definitions . . . 22
1.1 Decorated Context-Free Grammars . . . 22
1.2 Linear Context-Free Rewriting Systems . . . 23
1.3 Ranges . . . 24
2 Parsing as deduction . . . 25
2.1 Parsing decorated CFG . . . 25
3 The Naïve algorithm . . . 25
3.1 Inference rules . . . 26
4 The Approximative algorithm . . . 26
4.1 Inference rules . . . 28
5 The Active algorithm . . . 29

5.1 Inference rules . . . 29
6 The Incremental algorithm . . . 30
6.1 Inference rules . . . 30
7 Discussion . . . 31
7.1 Different prediction strategies . . . 32
7.2 Efficiency and complexity of the algorithms . . . 32
7.3 Implementing and testing the algorithms . . . 33

Paper 2: Natural Language Generation from Class Diagrams 37
1 Introduction . . . 38
1.1 Motivation . . . 39
1.2 Aim . . . 40
1.3 Contribution . . . 40
1.4 Overview . . . 40
2 Background . . . 41
2.1 Executable and Translatable UML . . . 41
2.2 Natural Language Generation . . . 42
2.3 Grammatical Framework . . . 43
3 Natural Language Generation from Class Diagrams . . . 45
3.1 Case Description . . . 45
3.2 xtUML to GF . . . 47
3.3 GF to Text . . . 49
4 Results . . . 49
5 Discussion . . . 50
6 Related Work . . . 50
7 Conclusions and Future Work . . . 52
7.1 Conclusion . . . 52
7.2 Future Work . . . 52

Paper 3: Executable and Translatable UML – How Difficult Can it Be? 57
1 Introduction . . . 58
1.1 Motivation . . . 59
1.2 Aim and Research Question . . . 60
1.3 Contribution . . . 60
1.4 Overview . . . 61
2 Executable and Translatable UML . . . 61
2.1 The Structure of xtUML . . . 61
2.2 Interpretation and Code Generation . . . 63
3 Case Study Design . . . 64
3.1 Subject and Case Selection . . . 64
3.2 Data Collection Procedures . . . 65
3.3 Evaluation Criteria . . . 66
4 Results . . . 66
4.1 Results from Evaluating the Models . . . 67
4.2 Outcomes From the Informal Discussions . . . 67
4.3 Experienced Learning Threshold . . . 68

4.4 Relevance to Industry 69
4.5 Evaluation of Validity 69
5 Discussion 71
6 Related Work 71
7 Conclusions and Future Work 72
7.1 Summary 72
7.2 Future Work 73

Introduction

1 Introduction

This thesis describes inter-disciplinary research in Language Technology and Software Engineering. The three included publications have a common theme in describing syntactical model transformations. Before we turn our attention towards transformations and our own research, we will first say a few words about Language Technology and Software Engineering.

1.1 Language Technology

The goal of Language Technology is to automatically process natural languages [17]. There are many examples of areas where this is useful: the spell checkers found in Microsoft Word and OpenOffice; machine translation, as by Google Translate but also as translation aids for professional translators; and extracting user response on the latest product release from Internet forums.

The spell checker needs a morphological analyser that can identify words and there different forms, such as plural forms for nouns, tense for verbs and conjugations for adjectives. It also needs a lexicon in order to suggest alternative spellings for unrecognised words.

A machine translator needs some kind of grammatical knowledge of the language, the syntax of the sentences. Questions should terminate with question marks and Swedish subordinate clauses have a different word order than full sentences. The machine translator also needs to know how constructs in the source language should be rendered in the target language.

The Internet is full of forums where customers and users discuss and voice their opinions about new technology. It is to expensive to employ people to monitor them all in order to see what is perceived as the pitfalls and benefits of a new release. The ability to automatically extract this information from free text and summarise it in predefined forms saves a lot of time and manual work, leading to shorter response times for updates and bug fixes. This requires a knowledge of the semantics and pragmatics of language to catch the meaning of each posting.

All these examples build on our possibility to model our own understanding of languages in a way readable by computers.

1.2 Software Engineering

Software surrounds us in our daily life. We have software in our cars, our phones, our cooking utensils and our washing machines. Our financial systems, our electricity distribution and international cargo transports all depend on software. Software Engineering is a discipline that focuses on how software can be specified, developed, verified and maintained [44].

Requirement specifications handle the expectations and limitations of software so that it is applicable and will be accepted by its users. The specifications have to be implemented into a working system through a development process and then verified and validated as to meeting the specifications and conditions. Hence testing has to be a part of Software Engineering. But just getting software to work is not enough, it is just as important to keep software working. Good software should enable upgrades and adaptation to changing requirements from users and changes in the contexts of the software.

1.3 Transformations

A central concept for both Language Technology and Software Engineering is the transformation. In Language Technology texts are analysed and transformed into internal representations that enable automatic analysis of the text. Or system-internal specifications are generated as text to enable more users to access the information. In Software Engineering transformations bring requirements into systems and enable existing systems to be updated and replaced.

1.4 Thesis Overview

The focus of the rest of this chapter is on transformations. In section 2 we give a more detailed account of transformations in the light of Language Technology and Software Engineering. We then relate the definitions from section 2 to our own research within the area of transformations in sections 3 to 5. These three sections have a shared structure; first the research is presented and put into context, then we describe the involved transformations and finally we give the scientific contribution and impact of each publication.

The included publications are:

Parsing Linear Context-Free Rewriting Systems A publication written together with Peter Ljunglöf from Computer Science and Engineering at Chalmers University of Technology and University of Gothenburg. Presented by Håkan Burden at the 9th International Workshop of Parsing Technologies, Vancouver, British Colombia, Canada in 2005 [11].

Natural Language Generation from Class Diagrams This publication was written together with Rogardt Heldal from Computer Science and Engineering at Chalmers University of Technology and University of Gothenburg. It was presented by Håkan Burden at the 8th MoDELS Workshop on Model-Driven Engineering, Verification and Validation, Wellington, New Zealand in 2011 [9].

Executable and Translatable UML – How Difficult Can it Be? This publication is joint work with Rogardt Heldal, Chalmers University of Technology and University of Gothenburg, and Toni Siljamäki, Ericsson. The publication was presented by Håkan Burden at the 18th Asia-Pacific Software Engineering Conference, Ho Chi Minh City, Vietnam in 2011 [10].

Reprints of the publications themselves are found in respective chapters. The intention is that the introductory sections in this chapter shall give some more background knowledge to each publication, without repeating what already is included in the publications themselves. As an example of the disposition, the grammar formalism Linear Context-Free Rewriting Systems (LCFRS) is defined in the publication and therefore the definition is not repeated in section 3, of the present chapter. Instead section 3 motivates the usage of LCFRS from a linguistic point of view. An exception from this setup is found in section 5 where we do not repeat the relevant concepts of software modelling that already have been introduced in section 4.

2 Transformations and Translations

Kleppe et al. [20] describe a transformation as a set of transformation rules that define how one or more constructs in the source language is mapped into one or more constructs of the target language. Mens and Van Gorp [31] suggest an addition to this definition in that transformations can have several input and output models. The definition given by Mellor et al. [30] supports this definition but they also stress that there has to be an algorithm for how to apply the transformation rules. In order to return syntactically valid models the transformation rules are defined in accordance to the grammar specifying the models [6, 30, 31], often referred to as the metamodel [33, 30]. In this way the transformations are not model-specific but apply to all models that conform to the same metamodel and are therefore reusable [30].

Mens and Van Gorp also state that a model transformation has two dimensions: a transformation between a source and a target language that share the same metamodel is endogenous, if the source and target have different metamodels the transformation is exogenous. Visser [52] refers to exogenous transformations as translations.

Vauquois [50] comes from a linguistic background and defines a translation as a series of transformations, see Figure 1. First, the source language is transformed into an intermediate representation according to the source language specification. The next step is a transfer from the source language's intermediate representation to the equivalent intermediate representation of the target language. Finally, the intermediate representation of the target language is used to generate the target language.

The transfers between the intermediate representations can be done at different levels. A direct transfer proceeds word-by-word through the source language and returns the corresponding word forms for the target language, in the same order. In this case the intermediate representation depends on a bilingual dictionary that analyses each word and returns the corresponding word for the target language. A syntactical transfer will use the syntactical knowledge of grammars to render the words of the target language in the right order. Examples of syntactical transfer rules would be to reorder the subject-verb-object structure of English to the subject-object-verb

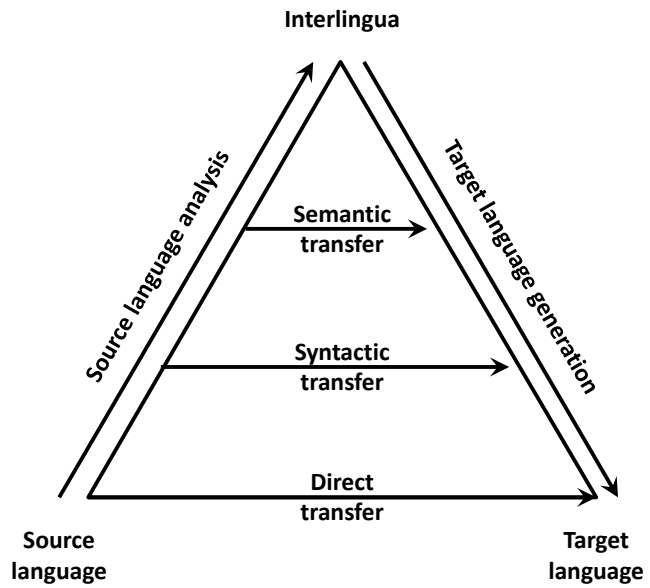

Figure 1: The Vauquois translation triangle

structure of Japanese or to adapt the noun-adjective order of Italian to the Swedish adjective-noun. A semantic transfer relies on an interpretation of the meaning of the source language as intermediate representation. This approach is often used for idiomatic expressions since their syntactic or direct transfer will often be meaningless in the target language.

A language specification that is shared by the source and target language is called an interlingua and eliminates the need for transferring between the intermediate representations. An interlingua translation is thus endogenous and not a translation, according to Visser.

Furthermore a transformation might add or remove information, making the target more abstract or more concrete than the source [31]. The exogenous transformations can then be categorised depending on how the translations change the level of abstraction:

Synthesis A translation from a more abstract source to a more concrete target language. The compilation of source code to machine code is a synthesis translation since a compiler will typically add information about hardware and operating systems, which does not have to be present in the source code [1, 31].

Reverse Engineering The opposite of synthesis. Source code can be used as a more concrete source for generating more abstract representations [12]. In this way

we can generate more abstract descriptions, such as use cases, for a system from its source code.

Migration Transforms the source language into a target language while preserving the level of abstraction. Translations of legal text, such as the proceedings of the European Parliament [21], have different specifications of the source and target languages while sharing the same level of abstraction. A special case of migration is when we combine synthesis with reverse engineering to get round trip engineering [23].

Transformations can be either unidirectional or bidirectional [31, 48]. Furthermore a transformation can be automatic or manual (also referred to as interactive by Stevens [48]).

3 Parsing Linear Context-Free Rewriting Systems

Before we describe the relationship between our own research on parsing and transformations in section 3.2 we need to introduce the main concepts, section 3.1. The contribution and impact of the parsing algorithms are then given in section 3.3.

3.1 Introduction

This publication presents four parsing algorithms for Linear Context-Free Rewriting Systems (LCFRS, [51]). At the time of publication there were no effective parsing algorithms available for LCFRS and the equivalent formalism Multiple-Context-Free Grammars (MCFG, [40]). This was a challenge since we saw an opportunity in using LCFRS for grammar development in an on-going research project [24, 25].

Linear Context-Free Rewriting Systems (LCFRS) are mildly context-sensitive [16] and can handle more complicated language structures than Context-Free Grammars [13]. In LCFRS a category, A, can be seen as returning a set of set of strings w;

$$A \Rightarrow^* \{\{w_{1_1}, \ldots, w_{1_m}\}, \ldots, \{w_{n_1}, \ldots, w_{n_m}\}\}$$

Since a category can yield a set of sets of strings, each individual set can span several substrings that are not adjacent, thus allowing multiple and crossed agreement as well as duplication [13, 16].

In Figure 2 there are two example sentences of subclauses with multiple and crossed agreement. The first sentence is a Swiss German subordinate clause with the corresponding English glosses[1] below. The subordinate clause can be translated into English as *"... we let the children help Hans paint the house"* [41]. The second example is in Dutch and translates as *"... that Jan saw Piet help Marie teach the children to swim"* [7]. The corresponding English glosses are given below the Dutch words. The arcs above the words show the dependencies between the nouns and the verbs. In the first example we get *em Hans* since *hälfe* requires the object to have dative case. In the second example the arcs shows who is doing what, i.e. *Jan* is *seeing* and *Marie* is

[1] The glosses can be seen as a direct transfer of the source language, see Figure 1.

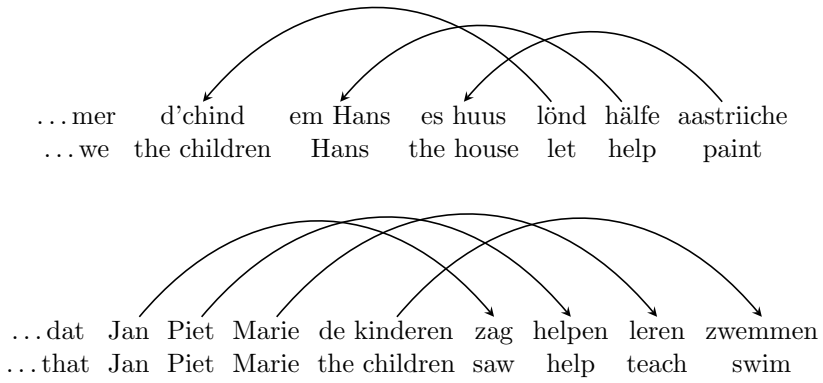

Figure 2: Multiple and crossed agreement in Swiss German and Dutch

teaching. Or in other words, an LCFRS grammar for Dutch can have a category that returns the set of sets of strings

$$\{\{\text{"}Jan\text{"}, \text{"}saw\text{"}\}, \{\text{"}Piet\text{"}, \text{"}help\text{"}\}, \{\text{"}Marie\text{"}, \text{"}teach\text{"}\}, \{\text{"}the\ children\text{"}, \text{"}swim\text{"}\} \ldots\}$$

3.2 Transformations

The four parsing algorithms are called Naïve, Approximative, Active and Incremental. All four parsing algorithms describe translations from text to a parse chart using the framework of parsing as deduction [42]. The transformation rules are described as deduction rules, using the grammar specification of LCFRS as our metamodel. The translation combines the input text with the subset of the grammar that describes the input into a parse chart. The chart will thus have a more concrete level of abstraction than the original source text.

3.2.1 Parsing as deduction

The idea behind parsing as deduction [42] is that parsing can be explained by deduction rules (also known as inference rules). A deduction rule can be written as

$$\frac{\begin{array}{c}\text{Antecedent}_1\\ \ldots\\ \text{Antecedent}_\text{n}\end{array}}{\text{Conclusion}}\{\text{Condition}$$

where the Conclusion is true if the Antecedents are true and the Conditions are fulfilled. A deduction rule without antecedents is called an axiom. All deductive systems need one or more axiomatic rule in order to introduce consequences to be used later on as antecedents.

```
Algorithm:       Agenda-driven chart parsing
Input:           A text and a grammar
Output:          Chart
Data structures: Chart, a set of deductions
                 Agenda, a set of deductions

for all axiomatic deduction rules
   deduce all consequences from conditions
   for each consequence
      if consequence not in chart
         add consequence to chart and agenda

while agenda contains consequences
   remove trigger from agenda
   deduce all consequences from trigger and chart
   for each consequence
      if consequence not in chart
         add consequence to chart and agenda

return chart
```

Figure 3: An agenda-driven chart parsing algorithm

As an example of parsing as deduction, lets consider a grammar rule for an English Sentence that consists of a Subject and a Predicate; Sentence → Subject Predicate. Under the condition of this rule we can deduce that we have a Sentence if there exists a Subject and a Predicate;

$$\frac{\begin{array}{c}\text{Subject}\\ \text{Predicate}\end{array}}{\text{Sentence}}\{\text{Sentence} \rightarrow \text{Subject Predicate}$$

The deduction algorithm can be implemented in many ways, one being as an agenda-driven algorithm, see Figure 3. Here the agenda keeps track of all the consequences that have not yet been used for deducing new consequences while we store all deduced consequences in a chart. Initially the agenda and the chart consist of the set of consequences deduced from the axiomatic rules.

We then remove one consequence at the time from the agenda, this consequence is referred to as the trigger and might trigger the deduction of new consequences in combination with consequences from the chart. The new consequences are added to both the chart and the agenda. We keep pulling new triggers until the agenda is empty. Finally, we return the chart that now contains the analysis of the input according to our grammar.

3.2.2 Parsing as a Transformation

In the context of Vauquois, Figure 1, parsing is equivalent to a syntactic analysis of the source language in a translation. Parsing a text is done according to a grammar, it is not possible to single out one and only one grammar that specifies the text; there might be many, there might be none and the parse chart will represent different analyses depending on which grammar that is used. This means that parsing is an endogenous transformation, a refinement, according to Mens and Van Gorp [31] that lowers the level of abstraction since the parse chart does not only contain the analysed parts of the input text with the according analysis, it also tells us what parts of the input we could not analyse.

3.2.3 Naïve

The Naïve algorithm is implemented in a bottom-up fashion, combining parse items representing smaller substrings of the text into items representing larger substrings. This is done by using three transformation rules. The algorithm got its name from the fact that it is a straight-forward application of context-free parsing techniques for LCFRS.

3.2.4 Approximative

The second algorithm, Approximative, got its name since it uses a context-free approximation of the LCFRS in the first of two transformations. The text is parsed by any chart-parsing algorithm using the (possibly over-generating) approximative context-free grammar. The context-free chart is then transformed into an LCFRS chart. The new chart items are combined bottom-up into new items in a way that is similar to how parsing is done in the Naïve algorithm. All in all the algorithm requires six deduction rules.

3.2.5 Active

In contrast to the previous algorithms, the Active algorithm relies on the set of possible strings of each category, instead of the categories themselves. The idea is to enumerate all strings of the set, adding new chart items whenever new information can be deduced from the inference rules. The deduction requires five different transformation rules. For this algorithm we proposed two filtering techniques adopted from context-free parsing, Earley [14] and Kilbury prediction [19]. The intention behind filtering is to limit the search space of the algorithm in order to get a more efficient run-time behaviour.

3.2.6 Incremental

The last algorithm is an adaptation of the Active algorithm. While the Active algorithm has full access to the text the Incremental algorithm reads the text once from left to right. Whenever a new word is read all possible consequences are computed before reading the next word. The transformation is described by four different deduction rules.

3.3 Contribution

The proposed filtering techniques for the Active algorithm were implemented and in the autumn of 2005 the Active algorithm with Kilbury filtering was the fastest. It resulted in a speedup of 20 times for English sentences, compared to the parsing algorithm that was used before our work. The algorithm was used for developing grammars in the EU-financed TALK-project [8, 27]. Since our publication Angelov [3] has improved the parsing of LCFRS and MCFG, both by an increase in efficiency but also by covering more complicated linguistic features. That work is also described within the framework of deductive parsing. Our work is the main publication used by Kallmeyer to describe LCFRS parsing in *Parsing Beyond Context-Free Grammars* [18].

4 Natural Language Generation from Class Diagrams

In this publication we describe natural language generation using an approach to software development called Model-Driven Architecture. The approach is realised by using tools for both Software Engineering and Language Technology, described in section 4.1. The two transformation steps are described in section 4.2 and their contribution in section 4.3.

4.1 Introduction

Software models are used to both analyse requirements and to specify the implementation of a system. Accessing the information of the models is not trivial, it requires an understanding of object-oriented design, knowledge of the used models and experience of using tools for software modelling in the development process [5]. These are skills that not all stakeholders might have. In contrast, natural language is understood by all stakeholders [15]. We decided to investigate the possibilities of transforming one type of software model, the class diagram, into natural language text. This was done in the context of Model-Driven Architecture, using Executable and Translatable UML to model the diagram and the Grammatical Framework for modelling the texts.

4.1.1 Model-Driven Architecture

In Model-Driven Architecture (MDA, [30, 33]) the Computationally Independent Model, CIM, typically includes descriptions of intended user interaction and the structure of the domain. These are formulated using natural languages and are open for interpretation. The CIM is then manually transformed into a Platform-Independent Model, PIM [44]. The PIM adds computational properties to the CIM, such as algorithms and interfaces. In this way the PIM is a bridge between the CIM and the Platform-Specific Model, PSM [35]. The PSM includes not only the behaviour and structure of the system, but also platform-specific details on how the PIM is to be realised in the context of operating systems, hardware, programming languages, tools and technologies for data-storage etc. In contrast to the PSM, the PIM can be reused to describe a

multitude of implementations [2]. The objective within MDA is that the PIM to PSM transformation should be automatic.

4.1.2 Executable and Translatable UML

One way of encoding the PIM is to use Executable and Translatable UML (xtUML, [29, 36, 47]) which is a graphical programming language. The abstraction level of xtUML is high enough to permit developers to design a PIM without having to consider platform-specific properties, while still having Turing complete expressivity [13]. The graphical models are executable and can be verified to deliver the expected functionality and structure [36] as well as translatable into efficient source code [43]. During the translation process platform-specific details are added in form of marks [29, 30]. For this project we used BridgePoint[2] to define the xtUML models.

4.1.3 Grammatical Framework

The Grammatical Framework (GF, [37]) is a Turing-complete grammar formalism [13]. The grammatical rules are described by an abstract syntax which is realised by one or more concrete syntaxes. All grammar rules have unique function names and are typed. An abstract rule can be written as $fun : Type$ where fun is the name of the rule and $Type$ is its type.

As a toy example we can have the two rules *fish_N : Noun* and *fish_V : Verb*, illustrating two disambiguations of the word *fish*. These abstract rules can now be implemented as concrete rules in the languages we want. For English we would have to have some structure corresponding to the type for nouns that enabled us to get the right word form depending on number; the plural form for *fish_N* returning *fish*. For verbs the type has to be more complex in order to correctly represent tense, person and number.

One of the benefits of GF is the Resource Grammar Library. The library covers 24 different languages[3], which are implemented by as many concrete grammars that share a common abstract syntax. The abstract syntax then works as an interlingua for bi-directional translation between the 22 languages (see Figure 1). By using the resource grammars we can define the concrete rules with the right types by *fish_N = mkN "fish" "fish"* and *fish_V = mkV "fish"* respectively, where the functions *mkN* and *mkV* are defined in the English resource grammar. We supply two arguments to *mkN* since *fish* has an irregular plural form. The resource grammar has more rules, that allow us to combine words and phrases into well-formed texts. The rules of the resource grammars raise the level of abstraction from the language-specific details to a more abstract level of syntactical descriptions.

4.2 Transformations

The transformation from class diagram to natural language texts was done in two steps. In the first step, the xtUML class diagram was automatically transformed

[2] `http://www.mentor.com/products/sm/model_development/bridgepoint/`
[3] `http://www.grammaticalframework.org/lib/doc/synopsis.html`

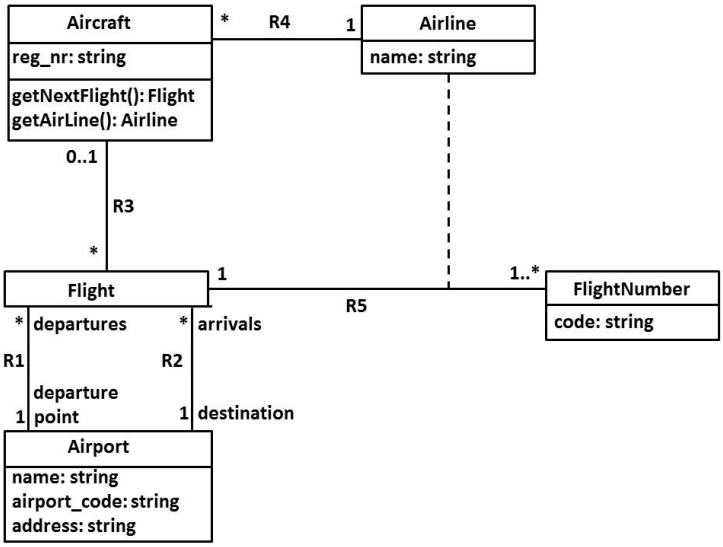

Figure 4: An xtUML class diagram

into a GF grammar. In the second step, the grammar was transformed into natural language text by linearisation.

The transformation rules of the first transformation are described by using the Rule Specification Language [32] which conforms to the BridgePoint metamodel for xtUML. The transformation is described by five major transformation rules that are applied top-down in the order they are specified. As a result of the transformation the outputted grammar and the class diagram share the same vocabulary but overall the transformation can be classified as reverse engineering [31] since not all information in the class diagram is carried over to the grammars. This transformation is both automatic and unidirectional with one input model and three output models; the abstract grammar, the concrete grammar and an abstract syntax tree that tells us in which order the grammatical rules shall be applied to generate our text. The syntactic correctness of the outputted grammars is guaranteed by the GF language specification [37].

In order to exemplify a model-to-grammar transformation we reuse the class diagram from the publication, Figure 4. We also need a metamodel for the diagram. For our purposes it is enough to assume that classes are referred to by `CLASS` in the metamodel and that they have the attribute `NAME`. Now `FlightNumber`, `Airport` and the other classes in Figure 4 are instances of `CLASS`. With a class diagram and a metamodel we can define a transfer rule, Figure 5, that returns an abstract and a concrete grammar for the class names in Figure 4.

Lines 1, 8 and 9 are comments which is shown by the row starting with the `.//` mark-up. On the second row we select all the instances of `CLASS` that can be found in the class diagram. We loop through all instances, lines 3–5, in order to output

```
01:  .// Generate abstract grammar rules
02:  .select many classes from instances of CLASS
03:  .for each class in classes
04:      ${class.NAME}C : N
05:  .end for
06:  .emit to file "AbstractClassNames.gf"
07:
08:  .// Generate concrete grammar rules using
09:  .// the English resource grammar
10:  .for each class in classes
11:      ${class.NAME}C = mkN "${class.NAME}"
12:  .end for
13:  .emit to file "ConcreteClassNames.gf"
```

Figure 5: An example of xtUML transformation rules

abstract grammar rules. Since row 4 is not begun with a dot it will render output every time it is triggered. By calling `emit` on row 13 the generated rules are written to the specified file. The procedure is repeated in lines 8-13 for the concrete grammar rules. As a result we get the abstract grammar `AbstractClassNames.gf` with rules like *AirportC : N* and the concrete grammar `ConcreteClassNames.gf` with rules on the form *AirportC = mkN "Airport"*.

The second transformation is also automatic and unidirectional but in contrast to the first transformation it is a synthesis translation from abstract syntax trees to natural language texts. The abstract syntax trees lack all information about the actual word forms and the word order of the generated text, this is stepwise introduced from the concrete syntax. The linearisation transformation is a part of the GF system and described in [4][4].

Overall the translation results in the class diagrams being reversed engineered into natural language texts.

4.3 Contribution

The result is a generic translation of any model that conforms to the BridgePoint metamodel. Since the model and the grammar share their vocabulary we can generate text for any domain, how technical it may be.

Overall the transformation from model to text follows the structure of Natural Language Generation (NLG; [38]). The first translation is equivalent to the text and sentence planning in NLG, the second transformation to the linguistic realisation.

This work is the first step towards generating textual descriptions automatically from the PIM, with the goal of covering the same information as the CIM. As a consequence the textual specifications, the PIM and the PSM can be synchronized and consistent with each other [22, 28, 49].

[4]We cite Ljunglöf [26] in our publication since Angelov's PhD thesis was not available at that time.

5 Executable and Translatable UML – How Difficult Can it Be?

This publication describes the effort for novice software modellers to transform a CIM defined by UML into a PIM defined by xtUML. Since both MDA and xtUML were introduced in section 4.1 these concepts are not introduced again. The manual transformation from CIM to PIM is described in 5.2 and the results from the case study are found in 5.3.

5.1 Introduction

We wanted to know how well bachelor students can handle the transformation from natural language requirements and analysis models, defined by using UML[5], to more concrete design models, defined by using xtUML. The effort lies both in understanding the transformation process but also in overcoming the learning threshold of xtUML as a specification language.

The two previous papers have in common that the authors were the once doing the transformations. This publication is different since students are doing the actual transformations while the authors monitor their activity. Monitoring the practice of others requires a more strict conduction of the study in order to gather the necessary information from the students without contaminating the validity of the findings. To ensure that this was done in a secure way the study primarily followed the recommendations of Runeson and Höst [39] and Yin [53].

5.2 Transformations

The translation as such was a manual transformation with multiple input models and one output model. It was manual since the automatic transformation of a CIM to a PIM is still a research area [44]. Due to the number of students and their different backgrounds there were no specific algorithm or rules for the translation. During the lectures we gave the students general guidelines how information from their CIM can be reused and transformed into a PIM. Larman [23] also gives some guidelines on transforming a CIM into a PIM when both are specified by UML. This text was also recommended to the students.

The xtUML metamodel is more allowing than we wanted. To narrow the scope of the target language it was not enough for the transformations to conform to the xtUML metamodel, we added our own criteria for a successful transformation. We encouraged the students to work incrementally by trying to get a small part working before adding new parts and we also specified what was most important to cover. Due to the variation in detail and the differences in functionality and structure as described by the CIM every translation to PIM was individual.

[5] `http://www.omg.org/spec/UML/2.4.1/`

5.3 Contribution

Over the two years, 43 out of 50 student teams succeeded in delivering verified and consistent models within the time frame. Due to the executable feature of the models the students were given constant feedback on their design until the models behaved as expected [36], with the required level of detail and structure. Since the time of publication another 24 translations have been carried out, with only one team failing in meeting our criteria. In total, 66 of 74 teams have successfully translated their UML CIMs into xtUML PIMs.

6 Future work

We want to continue our research on model-to-text transformations by further extending the scope of natural language generation from xtUML. The next step is then to generate texts from the behavioural model elements. Sridhara et al. [45, 46] have generated natural language descriptions from Java code. We aim to repeat their study but with a twist. Instead of reverse engineering the Java code into text we start from the more abstract Action Language of xtUML [29]. Since the abstraction level is higher from the beginning it should be easier to generate a text that avoids mentioning platform-specific details but instead focuses on the functionality itself.

We also want to further explore why xtUML is not used more. Earlier research show that the PIMs are reusable [2] and allow efficient code generation [43] while our publication shows that undergraduate students cope with the translation from a CIM defined by UML to a PIM confirming to the xtUML metamodel. Drawing on our ongoing industrial collaboration we want to investigate the industrial practice of xtUML and what software engineers find as advantages and drawbacks of xtUML. This line of future work ties in with the first track, since generating natural language descriptions can include more stakeholders into the development process and make xtUML a more applicable technology.

7 Conclusion

We have described three syntactical transformations.

The first transformation describes four parsing algorithms that takes a text as input and returns its analysis according to a grammar. The transformation is automatic and endogenous since the text and analysis use the same grammar as specification and the output has a lower level of abstraction than the input. Parsing is to its nature unidirectional, but the underlying algorithm can vary between different parsing approaches. In our case the transformation algorithm is described as parsing by deduction.

While parsing is endogenous, natural language generation is exogenous. The unidirectional transformation from class diagram to text is done in two steps. In the first step the diagram is automatically transformed into three output models; the abstract grammar, the concrete grammar and an abstract syntax tree. The syntax tree describes how the grammars are to be used in the second transformation step to yield the desired text. Overall the transformation is an example of reverse engineering.

The third publication describes how well novice software modellers managed to manually transform a set of UML models into xtUML models. The outcome has a lower level of abstraction than the input, and serve as an example of a unidirectional synthesis translation. The translations do not follow a clearly defined algorithm.

The transformations are conducted within the disciplines of Language Technology and Software Engineering. The generation of natural language texts from software models is in fact the result of combining tools and technologies from both fields. We see ample possibilities for continuing our research in combining the strengths and possibilities of respective area.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education, Inc., Boston, 2007.

[2] Staffan Andersson and Toni 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.

[3] Krasimir Angelov. Incremental Parsing of Parallel Multiple Context-Free Grammars. In *12th Conference of the European Chapter of the Association for Computational Linguistics*, 2009.

[4] Krasimir Angelov. *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University Of Technology, Gothenburg, Sweden, 2011.

[5] Jim Arlow, Wolfgang Emmerich, and John 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.

[6] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36 – 41, sept.-oct. 2003.

[7] Joan W. Bresnan, Ronald M. Kaplan, P. Stanley Peters, and Annie Zaenen. Cross-serial Dependencies in Dutch. *Linguistic Inquiry*, 13:613–635, 1982.

[8] Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, pages 53–60, June 2005.

[9] Håkan Burden and Rogardt Heldal. Natural Language Generation from Class Diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa 2011, Wellington, New Zealand, October 2011. ACM.

[10] Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML – How Difficult Can it Be? In *APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

[11] Håkan Burden and Peter Ljunglöf. Parsing linear context-free rewriting systems. In *IWPT'05, 9th International Workshop on Parsing Technologies*, Vancouver, BC, Canada, 2005.

[12] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[13] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[14] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[15] Donald Firesmith. Modern Requirements Specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[16] Aravind Joshi. How Much Context-Sensitivity is Necessary for Characterizing Structural Descriptions — Tree Adjoining Grammars. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Processing: Psycholinguistic, Computational and Theoretical Perspectives*, pages 206–250. Cambridge University Press, New York, 1985.

[17] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Pearson Education Inc., Upper Saddle River, New Jersey, USA, 2 edition, 2009.

[18] Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. Springer, 2010.

[19] James Kilbury. Chart parsing and the Earley algorithm. In Ursula Klenk, editor, *Kontextfreie Syntaxen und wervandte Systeme*. Niemeyer, Tübingen, Germany, 1985.

[20] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, 2005.

[21] Philipp Koehn. Europarl: A Parallel Corpus for Statistical Machine Translation. In *Conference Proceedings: the 10th Machine Translation Summit*, pages 79–86, Phuket, Thailand, 2005. Asia-Pacific Association for Machine Translation.

[22] Christian F. J. Lange and Michel R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 401–411, New York, NY, USA, 2006. ACM.

[23] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[24] Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University and Chalmers University of Technology, November 2004.

[25] Peter Ljunglöf. Grammatical Framework and Multiple Context-Free Grammars. In *9th Conference on Formal Grammar*, Nancy, France, 2004.

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

[27] Peter Ljunglöf, Björn Bringert, Robin Cooper, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson, and Aarne Ranta. The TALK grammar library: an integration of GF with TrindiKit. Deliverable D1.1, TALK Project, 2005.

[28] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

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

[30] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[31] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[32] Mentor Graphics. *BridgePoint UML Suite Rule Specification Language*.

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

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

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

[36] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

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

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

[39] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[40] Hiroyuki Seki, Takashi Matsumara, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

[41] Stuart Shieber. Evidence against the context-freeness of natural language. *Computational Linguistics*, 20(2):173–192, 1985.

[42] Stuart Shieber, Yves Schabes, and Fernando Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

[43] Toni Siljamäki and Staffan Andersson. Performance benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010.

[45] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori 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.

[46] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 101–110, New York, NY, USA, 2011. ACM.

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

[48] Perdita Stevens. A landscape of bidirectional model transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2007.

[49] Ragnhild Van Der Straeten. Description of UML Model Inconsistencies. Technical report, Software Languages Lab, Vrije Universiteit Brussel, 2011.

[50] Bernard Vauquois. A survey of formal grammars and algorithms for recognition and transformation in mechanical translation. In *Information Processing 68, Proceedings of IFIP Congress 1968*, 2, pages 1114–1122, 1968.

[51] K. Vijay-Shanker, David Weir, and Aravind Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Meeting of the Association for Computational Linguistics*, 1987.

[52] Eelco Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[53] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, California, fourth edition, 2009.

Paper 1

Parsing Linear Context-Free Rewriting Systems

Reprint from the proceedings of:

IWPT'05
9th International Workshop on Parsing Technologies

Vancouver, BC, Canada
October 2005

Parsing Linear Context-Free Rewriting Systems

Håkan Burden[1] and Peter Ljungöf[2]

[1] Dept. of Linguistics, University of Gothenburg, Göteborg, Sweden
cl1hburd@cling.gu.se

[2] Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden
peb@chalmers.se

Abstract

We describe four different parsing algorithms for Linear Context-Free Rewriting Systems [11]. The algorithms are described as deduction systems, and possible optimizations are discussed.

The only parsing algorithms presented for *linear context-free rewriting systems* (LCFRS; Vijay-Shanker et al., 1987) and the equivalent formalism *multiple context-free grammar* (MCFG; Seki et al., 1991) are extensions of the CKY algorithm [13], more designed for their theoretical interest, and not for practical purposes. The reason for this could be that there are not many implementations of these grammar formalisms. However, since a very important subclass of the Grammatical Framework [7] is equivalent to LCFRS/MCFG [4, 5], there is a need for practical parsing algorithms.

In this paper we describe four different parsing algorithms for Linear Context-Free Rewriting Systems. The algorithms are described as deduction systems, and possible optimizations are discussed.

1 Introductory definitions

A *record* is a structure $\Gamma = \{r_1 = a_1; \ldots; r_n = a_n\}$, where all r_i are distinct. That this can be seen as a set of feature-value pairs. This means that we can define a simple version of *record unification* $\Gamma_1 \sqcup \Gamma_2$ as the union $\Gamma_1 \cup \Gamma_2$, provided that there is no r such that $\Gamma_1.r \neq \Gamma_2.r$.

We sometimes denote a sequence $X_1, \ldots, X_n$ by the more compact $\vec{X}$. To update the ith record in a list of records, we write $\vec{\Gamma}[i := \Gamma]$. To substitute a variable B_k for a record Γ_k in any data structure Γ, we write $\Gamma[B_k/\Gamma_k]$.

1.1 Decorated Context-Free Grammars

The context-free approximation described in section 4 uses a form of CFG with decorated rules of the form $f : A \to \alpha$, where f is the name of the rule, and α is a sequence of terminals and categories subscripted with information needed for post-processing of the context-free parse result. In all other respects a decorated CFG can be seen as a straight-forward CFG.

$$
\begin{aligned}
S \to f[A] &:= \{\, s = A.p\ A.q \,\} \\
A \to g[A_1\ A_2] &:= \{\, p = A_1.p\ A_2.p;\ q = A_1.q\ A_2.q \,\} \\
A \to ac[\,] &:= \{\, p = a;\ q = c \,\} \\
A \to bd[\,] &:= \{\, p = b;\ q = d \,\}
\end{aligned}
$$

Figure 1: An example grammar describing the language

1.2 Linear Context-Free Rewriting Systems

A *linear context-free rewriting system* (LCFRS; Vijay-Shanker et al., 1987) is a linear, non-erasing *multiple context-free grammar* (MCFG; Seki et al., 1991). An MCFG rule is written[1]

$$A \to f[B_1 \ldots B_\delta] \quad := \quad \{\, r_1 = \alpha_1;\ \ldots;\ r_n = \alpha_n \,\}$$

where A and B_i are categories, f is the name of the rule, r_i are record labels and α_i are sequences of terminals and argument projections of the form $B_i.r$. The *language* $\mathcal{L}(A)$ of a category A is a set of string records, and is defined recursively as

$$
\begin{aligned}
\mathcal{L}(A) \quad = \quad \{ \quad & \Phi[B_1/\Gamma_1, \ldots, B_\delta/\Gamma_\delta] \quad | \\
& A \to f[B_1 \ldots B_\delta] := \Phi, \\
& \Gamma_1 \in \mathcal{L}(B_1),\ \ldots,\ \Gamma_\delta \in \mathcal{L}(B_\delta) \quad \}
\end{aligned}
$$

It is the possibility of discontinuous constituents that makes LCFRS/MCFG more expressive than context-free grammars. If the grammar only consists of single-label records, it generates a context-free language.

Example A small example grammar is shown in figure 1, and generates the language

$$\mathcal{L}(S) \quad = \quad \{\ s\, s_{hm} \mid s \in (a \cup b)^*\ \}$$

where s_{hm} is the homomorphic mapping such that each a in s is translated to c, and each b is translated to d. Examples of generated strings are *ac*, *abcd* and *bbaddc*. However, neither *abc* nor *abcdabcd* will be generated. The language is not context-free since it contains a combination of multiple and crossed agreement with duplication.

If there is at most one occurrence of each possible projection $A_i.r$ in a linearization record, the MCFG rule is *linear*. If all rules are linear the grammar is linear. A rule is *erasing* if there are argument projections that have no realization in the linearization. A grammar is erasing if it contains an erasing rule. It is possible to transform an erasing grammar to non-erasing form [8].

[1] We borrow the idea of equating argument categories and variables from Nakanishi et al. [6] , but instead of tuples we use the equivalent notion of records for the linearizations.

Example The example grammar is both linear and non-erasing. However, given that grammar, the rule

$$E \to e[A] \quad := \quad \{\, r_1 = A.p;\, r_2 = A.p \,\}$$

is both non-linear (since $A.p$ occurs more than once) and erasing (since it does not mention $A.q$).

1.3 Ranges

Given an input string w, a *range* ρ is a pair of indices, (i, j) where $0 \leq i \leq j \leq |w|$ [1]. The entire string $w = w_1 \ldots w_n$ spans the range $(0, n)$. The word w_i spans the range $(i-1, i)$ and the substring $w_{i+1}, \ldots, w_j$ spans the range (i, j). A range with identical indices, (i, i), is called an empty range and spans the empty string.

A record containing label-range pairs,

$$\Gamma \quad = \quad \{\, r_1 = \rho_1, \ldots, r_n = \rho_n \,\}$$

is called a *range record*. Given a range $\rho = (i, j)$, the *ceiling* of ρ returns an empty range for the right index, $\lceil \rho \rceil = (j, j)$; and the *floor* of ρ does the same for the left index $\lfloor \rho \rfloor = (i, i)$. Concatenation of two ranges is non-deterministic,

$$(i, j) \cdot (j', k) \quad = \quad \{\, (i, k) \,|\, j = j' \,\}$$

.

1.3.1 Range restriction

In order to retrieve the ranges of any substring s in a sentence $w = w_1 \ldots w_n$ we define *range restriction* of s with respect to w as $\langle s \rangle^w = \{\, (i, j) \,|\, s = w_{i+1} \ldots\, w_j \,\}$, i.e. the set of all occurrences of s in w. If w is understood from the context we simply write $\langle s \rangle$.

Range restriction of a linearization record Φ is written $\langle \Phi \rangle$, which is a set of records, where every terminal token s is replaced by a range from $\langle s \rangle$. The range restriction of two terminals next to each other fails if range concatenation fails for the resulting ranges. Any unbound variables in Φ are unaffected by range restriction.

Example Given the string $w = abba$, range restricting the terminal a yields

$$\langle a \rangle^w \quad = \quad \{\, (0, 1),\, (3, 4) \,\}$$

Furthermore,

$$\langle a\, A.r\, a\, b\, B.q \rangle^w \quad = \\ \{\, (0, 1)\, A.r\, (0, 2)\, B.q,\ (3, 4)\, A.r\, (0, 2)\, B.q \,\}$$

The other possible solutions fail since they cannot be range concatenated.

2 Parsing as deduction

The idea with *parsing as deduction* [9] is to deduce parse items by inference rules. A parse item is a representation of a piece of information that the parsing algorithm has acquired. An inference rule is written

$$\frac{\begin{array}{c}\gamma_1 \ldots \gamma_n \\ C\end{array}}{\gamma}$$

where γ is the consequence of the antecedents $\gamma_1 \ldots \gamma_n$, given that the side conditions in C hold.

2.1 Parsing decorated CFG

Decorated CFG can be parsed in a similar way as standard CFG. For our purposes it suffices to say that the algorithm returns items of the form,

$$[f : A/\rho \rightarrow B_1/\rho_1 \ \ldots \ B_n/\rho_n \bullet]$$

saying that A spans the range ρ, and each daughter B_i spans ρ_i.

The standard inference rule *combine* might look like this for decorated CFG:

Combine

$$\frac{\begin{array}{l}[f : A/\rho \rightarrow \alpha \bullet B_x \, \beta] \\ [g : B/\rho' \rightarrow \ldots \bullet] \\ \rho'' \in \rho \cdot \rho'\end{array}}{[f : A/\rho \rightarrow \alpha \, B_x/\rho'' \bullet \beta]}$$

Note that the subscript x in B_x is the decoration that will only be used in post-processing.

3 The Naïve algorithm

Seki et al. [8] give an algorithm for MCFG, which can be seen as an extension of the CKY algorithm [13]. The problem with that algorithm is that it has to find items for all daughters at the same time. We modify this basic algorithm to be able to find one daughter at the time.

There are two kinds of items. A *passive* item $[A; \Gamma]$ has the meaning that the category A has been found spanning the range record Γ. An *active* item for the rule $A \rightarrow f[\vec{B} \, \vec{B}'] := \Psi$ has the form

$$[A \rightarrow f[\vec{B} \bullet \vec{B}'];\ \Phi;\ \vec{\Gamma}]$$

in which the categories to the left of the dot, $\vec{B}$, have been found with the linearizations in the list of range records $\vec{\Gamma}$. Φ is the result of substituting the projections in Ψ with ranges for the categories found in $\vec{B}$.

3.1 Inference rules

There are three inference rules, Predict, Combine and Convert.

Predict

$$\frac{\begin{array}{l} A \to f[\vec{B}] := \Psi \\ \Phi \in \langle \Psi \rangle \end{array}}{[A \to f[\bullet \vec{B}];\ \Phi;\ ]}$$

Prediction gives an item for every rule in the grammar, where the range restriction Φ is what has been found from the beginning. The list of daughters is empty since none of the daughters in $\vec{B}$ have been found yet.

Combine

$$\frac{\begin{array}{l} [A \to f[\vec{B} \bullet B_k\, \vec{B'}];\ \Phi;\ \vec{\Gamma}] \\ [B_k;\ \Gamma_k] \\ \quad \Phi' \in \Phi[B_k/\Gamma_k] \end{array}}{[A \to f[\vec{B}\, B_k \bullet \vec{B'}];\ \Phi';\ \vec{\Gamma},\ \Gamma_k]}$$

An active item looking for B_k and a passive item that has found B_k can be combined into a new active item. In the new item we substitute B_k for Γ_k in the linearization record. We also add Γ_k to the new item's list of daughters.

Convert

$$\frac{\begin{array}{l} [A \to f[\vec{B} \bullet];\ \Phi;\ \vec{\Gamma}] \\ \Gamma \equiv \Phi \end{array}}{[A;\ \Gamma]}$$

Every fully instantiated active item is converted into a passive item. Since the linearization record Φ is fully instantiated, it is equivalent to the range record Γ.

4 The Approximative algorithm

Parsing is performed in two steps in the approximative algorithm. First we parse the sentence using a context-free approximation. Then the resulting context-free chart is recovered to a LCFRS chart.

The LCFRS is converted by creating a decorated context-free rule for every row in a linearization record. Thus, the rule

$$A \to f[\vec{B}] \quad := \quad \{\, r_1 = \alpha_1;\ \ldots;\ r_n = \alpha_n \,\}$$

will give n context-free rules $f : A.r_i \to \alpha_i$. The example grammar from figure 1 is converted to a decorated CFG in figure 2.

$$
\begin{array}{rcl}
f : (S.s) & \rightarrow & (A.p)\ (A.q) \\
g : (A.p) & \rightarrow & (A.p)_1\ (A.p)_2 \\
g : (A.q) & \rightarrow & (A.q)_1\ (A.q)_2 \\
ac : (A.p) & \rightarrow & a \\
ac : (A.q) & \rightarrow & b \\
bd : (A.p) & \rightarrow & c \\
bd : (A.q) & \rightarrow & d
\end{array}
$$

The subscripted numbers are for distinguishing the two categories from each other, since they are equivalent. Here $A.q$ is a context-free category of its own, not a record projection.

Figure 2: The example grammar converted to a decorated CFG

Parsing is now initiated by a context-free parsing algorithm returning decorated items as in section 2.1. Since the categories of the decorated grammar are projections of LCFRS categories, the final items will be of the form

$$[f : (A.r)/\rho \rightarrow \ldots (B.r')_x/\rho' \ldots \bullet\,]$$

Since the decorated CFG is over-generating, the returned parse chart is unsound. We therefore need to retrieve the items from the decorated CFG parse chart and check them against the LCFRS to get the discontinuous constituents and mark them for validity.

The *initial parse items* are of the form,

$$[A \rightarrow f[\vec{B}];\, r = \rho;\, \vec{\Gamma}]$$

where $\vec{\Gamma}$ is extracted from a corresponding decorated item $[f : (A.r)/\rho \rightarrow \beta]$, by partitioning the daughters in β such that $\Gamma_i = \{\, r = \rho \,|\, (B.r)_i/\rho \in \beta \,\}$. In other words, Γ_i will consist of all $r = \rho$ such that $B.r$ is subscripted by i in the decorated item.

Example Given $\beta = (A.p)_2/\rho'\ (B.q)_1/\rho''\ (A.q)_2/\rho'''$, we get the two range records $\Gamma_1 = \{q = \rho''\}$ and $\Gamma_2 = \{p = \rho';\, q = \rho'''\}$.

Apart from the initial items, we use three kinds of parse items. From the initial parse items we first build *LCFRS items*, of the form

$$[A \rightarrow f[\vec{B}];\, \Gamma \bullet r_i \ldots r_n;\, \vec{\Gamma}]$$

where $r_i \ldots r_n$ is a list of labels, $\vec{\Gamma}$ is a list of $|\vec{B}|$ range records, and Γ is a range record for the labels $r_1 \ldots r_{i-1}$.

In order to recover the chart we use *mark items*

$$[A \rightarrow f[\vec{B} \bullet \vec{B}'];\, \Gamma;\, \vec{\Gamma} \bullet \vec{\Gamma}']$$

The idea is that $\vec{\Gamma}$ has been verified as range records spanning the daughters $\vec{B}$. When all daughters have been verified, a mark item is converted to a *passive item* $[A;\, \Gamma]$.

4.1 Inference rules

There are five inference rules, Pre-Predict, Pre-Combine, Mark-Predict, Mark-Combine and Convert.

Pre-Predict

$$\frac{\begin{array}{l} A \to f[\vec{B}] := \{r_1 = \alpha_1; \ldots; r_n = \alpha_n\} \\ \vec{\Gamma}_\delta = \{\,\}, \ldots, \{\,\} \end{array}}{[A \to f[\vec{B}];\ \bullet\, r_1 \ldots r_n;\ \vec{\Gamma}_\delta]}$$

Every rule $A \to f[\vec{B}]$ is predicted as an LCFRS item. Since the context-free items contain information about $\alpha_1 \ldots \alpha_n$, we only need to use the labels $r_1, \ldots, r_n$. $\vec{\Gamma}_\delta$ is a list of $|\vec{B}|$ empty range records.

Pre-Combine

$$\frac{\begin{array}{l} [R;\ \Gamma \bullet r\, r_i \ldots r_n;\ \vec{\Gamma}] \\ [R;\ r = \rho;\ \vec{\Gamma}'] \\ \vec{\Gamma}'' \in \vec{\Gamma} \sqcup \vec{\Gamma}' \end{array}}{[R;\ \{\Gamma; r = \rho\} \bullet r_i \ldots r_n;\ \vec{\Gamma}'']}$$

If there is an initial parse item for the rule R with label r, we can combine it with an LCFRS item looking for r, provided the daughters' range records can be unified.

Mark-Predict

$$\frac{[A \to [\vec{B}];\ \Gamma \bullet;\ \vec{\Gamma}]}{[A \to [\,\bullet\, \vec{B}];\ \Gamma;\ \bullet\, \vec{\Gamma}]}$$

When all record labels have been found, we can start to check if the items have been derived in a valid way by marking the daughters' range records for correctness.

Mark-Combine

$$\frac{\begin{array}{l} [A \to f[\vec{B} \bullet B_i\, \vec{B}'];\ \Gamma;\ \vec{\Gamma} \bullet \Gamma_i\, \vec{\Gamma}'] \\ [B_i;\ \Gamma_i] \end{array}}{[A \to f[\vec{B}\, B_i \bullet \vec{B}'];\ \Gamma;\ \vec{\Gamma}\, \Gamma_i \bullet \vec{\Gamma}']}$$

Record Γ_i is correct if there is a correct passive item for category B_i that has found Γ_i.

Convert

$$\frac{[A \to f[\vec{B} \bullet];\ \Gamma;\ \vec{\Gamma} \bullet]}{[A;\ \Gamma]}$$

An item that has marked all daughters as correct is converted to a passive item.

5 The Active algorithm

The active algorithm parses without using any context-free approximation. Compared to the Naïve algorithm the dot is used to traverse the linearization record of a rule instead of the categories in the right-hand side.

For this algorithm we use a special kind of range, ρ^ϵ, which denotes simultaneously all empty ranges (i, i). Range restricting the empty string gives $\langle\epsilon\rangle = \rho^\epsilon$. Concatenation is defined as $\rho \cdot \rho^\epsilon = \rho^\epsilon \cdot \rho = \rho$. Both the ceiling and the floor of ρ^ϵ are identities, $\lceil\rho^\epsilon\rceil = \lfloor\rho^\epsilon\rfloor = \rho^\epsilon$.

There are two kinds of items. *Passive items* $[A; \Gamma]$ say that we have found category A inside the range record Γ. An *active item* for the rule

$$A \to f[\vec{B}] \quad := \quad \{\Phi;\, r = \alpha\beta;\, \Psi\}$$

is of the form

$$[A \to f[\vec{B}];\, \Gamma,\, r = \rho \bullet \beta,\, \Psi;\, \vec{\Gamma}]$$

where Γ is a range record corresponding to the linearization rows in Φ and α has been recognized spanning ρ. We are still looking for the rest of the row, β, and the remaining linearization rows Ψ. $\vec{\Gamma}$ is a list of range records containing information about the daughters $\vec{B}$.

5.1 Inference rules

There are five inference rules, Predict, Complete, Scan, Combine and Convert.

Predict

$$\frac{\begin{array}{l} A \to f[\vec{B}] := \{r = \alpha; \Phi\} \\ \vec{\Gamma}_\delta = \{\,\}, \ldots, \{\,\} \end{array}}{[A \to f[\vec{B}];\, \{\},\, r = \rho^\epsilon \bullet \alpha,\, \Phi;\, \vec{\Gamma}_\delta]}$$

For every rule in the grammar, predict a corresponding item that has found the empty range. $\vec{\Gamma}_\delta$ is a list of $|\vec{B}|$ empty range records since nothing has been found yet.

Complete

$$\frac{[R;\, \Gamma,\, r = \rho \bullet \epsilon,\, \{r' = \alpha; \Phi\};\, \vec{\Gamma}]}{[R;\, \{\Gamma; r = \rho\},\, r' = \rho^\epsilon \bullet \alpha, \Phi;\, \vec{\Gamma}]}$$

When an item has found an entire linearization row we continue with the next row by starting it off with the empty range.

Scan

$$\frac{\begin{array}{l} [R;\, \Gamma,\, r = \rho \bullet s\,\alpha,\, \Phi;\, \vec{\Gamma}] \\ \rho' \in \rho \cdot \langle s \rangle \end{array}}{[R;\, \Gamma,\, r = \rho' \bullet \alpha,\, \Phi;\, \vec{\Gamma}]}$$

When the next symbol to read is a terminal, its range restriction is concatenated with the range for what the row has found so far.

Combine

$$\frac{\begin{array}{l}[A \to f[\vec{B}];\, \Gamma,\, r = \rho \bullet B_i.r'\,\alpha,\, \Phi;\, \vec{\Gamma}] \\ [B_i;\, \Gamma'] \\ \rho' \in \rho \cdot \Gamma'.r' \\ \Gamma_i \subseteq \Gamma'\end{array}}{[A \to f[\vec{B}];\, \Gamma,\, r = \rho' \bullet \alpha,\, \Phi;\, \vec{\Gamma}[i := \Gamma']]}$$

If the next thing to find is a projection on B_i, and there is a passive item where B_i is the category, where Γ' is consistent with Γ_i, we can move the dot past the projection. Γ_i is updated with Γ', since it might contain more information about the ith daughter.

Convert

$$\frac{[A \to f[\vec{B}];\, \Gamma,\, r = \rho \bullet \epsilon,\, \{\};\, \vec{\Gamma}]}{[A;\, \{\Gamma; r = \rho\}]}$$

An active item that has fully recognized all its linearization rows is converted to a passive item.

6 The Incremental algorithm

An incremental algorithm reads one token at the time and calculates all possible consequences of the token before the next token is read[2]. The Active algorithm as described above is not incremental, since we do not know in which order the linearization rows of a rule are recognized. To be able to parse incrementally, we have to treat the linearization records as sets of feature-value pairs, instead of a sequence.

The items for a rule $A \to f[\vec{B}] := \Phi$ have the same form as in the Active algorithm:

$$[A \to f[\vec{B}];\, \Gamma,\, r = \rho \bullet \beta,\, \Psi;\, \vec{\Gamma}]$$

However, the order between the linearization rows does not have to be the same as in Φ. Note that in this algorithm we do not use passive items. Also note that since we always know where in the input we are, we cannot make use of a distinguished ϵ-range. Another consequence of knowing the current input position is that there are fewer possible matches for the Combine rule.

6.1 Inference rules

There are four inference rules, Predict, Complete, Scan and Combine.

[2]See e.g. the ACL 2004 workshop "Incremental Parsing: Bringing Engineering and Cognition Together".

Predict

$$\frac{\begin{array}{l} A \to f[\vec{B}] := \{\Phi; r = \alpha; \Psi\} \\ 0 \leq k \leq |w| \end{array}}{[A \to f[\vec{B}];\ \{\},\ r = (k,k) \bullet \alpha,\ \{\Phi; \Psi\};\ \vec{\Gamma}_\delta]}$$

An item is predicted for every linearization row r and every input position k. $\vec{\Gamma}_\delta$ is a list of $|\vec{B}|$ empty range records.

Complete

$$\frac{\begin{array}{l} [R;\ \Gamma,\ r = \rho \bullet \epsilon,\ \{\Phi; r' = \alpha; \Psi\};\ \vec{\Gamma}] \\ \lceil \rho \rceil \leq k \leq |w| \end{array}}{[R;\ \{\Gamma; r = \rho\},\ r' = (k,k) \bullet \alpha,\ \{\Phi; \Psi\};\ \vec{\Gamma}]}$$

Whenever a linearization row r is fully traversed, we predict an item for every remaining linearization row r' and every remaining input position k.

Scan

$$\frac{\begin{array}{l} [R;\ \Gamma,\ r = \rho \bullet s\,\alpha,\ \Phi;\ \vec{\Gamma}] \\ \rho' \in \rho \cdot \langle s \rangle \end{array}}{[R;\ \Gamma,\ r = \rho' \bullet \alpha,\ \Phi;\ \vec{\Gamma}]}$$

If the next symbol in the linearization row is a terminal, its range restriction is concatenated to the range for the partially recognized row.

Combine

$$\frac{\begin{array}{l} [R; \Gamma, r = \rho \bullet B_i.r'\,\alpha, \Phi; \vec{\Gamma}] \\ [B_i \to \ldots;\ \Gamma',\ r' = \rho' \bullet \epsilon,\ \ldots;\ \ldots] \\ \rho'' \in \rho \cdot \rho' \\ \Gamma_i \subseteq \{\Gamma'; r' = \rho'\} \end{array}}{[R; \Gamma, r = \rho'' \bullet \alpha, \Phi; \vec{\Gamma}[i := \{\Gamma'; r' = \rho'\}]]}$$

If the next item is a record projection $B_i.r'$, and there is an item for B_i which has found r', then move the dot forward. The information in Γ_i must be consistent with the information found for the B_i item, $\{\Gamma'; r' = \rho'\}$.

7 Discussion

We have presented four different parsing algorithms for LCFRS/MCFG. The algorithms are described as deduction systems, and in this final section we discuss some possible optimizations, and complexity issues.

7.1 Different prediction strategies

The Predict rule in the above described algorithms is very crude, predicting an item for each rule in the grammar (for the Incremental algorithm even for each input position). A similar context-free prediction rule is called *bottom-up Earley* by Sikkel and Nijholt [10]. Such crude predictions are only intended for educational purposes, since they lead to lots of uninteresting items, and waste of computing power. For practical purposes there are two standard context-free prediction strategies, top-down and bottom-up (see e.g. Wirén [12]) and they can be adapted to the algorithms presented in this paper.

The main idea is that an item for the rule $A \to f[\vec{B}]$ with the linearization row $r = \alpha$ is only predicted if...

(Top-down prediction) ...there is another item looking for $A.r$.

(Bottom-up prediction) ...there is an passive item that has found the first symbol in α.

For a more detailed description of these prediction strategies, see Ljunglöf [4].

7.2 Efficiency and complexity of the algorithms

The theoretical time complexity for these algorithms is not better than what has been presented earlier.[3] The complexity arguments are similar and the reader is referred to Seki et al. [8].

However, theoretical time complexity does not say much about practical performance, as is already clear from context-free parsing, where the theoretical time complexity has remained the same ever since the first publications [3, 13]. There are two main ways of improving the efficiency of existing algorithms, which can be called *refinement* and *filtering* [10]. First, one wants to be able to locate existing parse items efficiently, e.g. by indexing some properties in a hash table. This is often done by *refining* the parse items or inference rules, increasing the number of items or deduction steps. Second, it is desirable to reduce the number of parse items, which can be done by *filtering* out redundant parts of an algorithm.

The algorithms presented in this paper can all be seen as refinements and filterings of the basic algorithm of Seki et al. [8]:

The naïve algorithm is a refinement of the basic algorithm, since single items and deduction steps are decomposed into several different items and smaller deduction steps.

The approximative algorithm is both a refinement and a filtering of the naïve algorithm; a refinement since the inference rules Pre-Predict and Pre-Combine are added, and a filtering since there will hopefully be less items for Mark-Predict and Mark-Combine to take care of.

[3] Nakanishi et al. [6] reduce the parsing problem to boolean matrix multiplication, but this can be considered a purely theoretical result.

The active algorithm is a refinement of the naïve algorithm, since the Combine rule is divided into the rules Complete, Scan and Combine.

The incremental algorithm is finally a refinement of the active algorithm, since Predict and Complete can select from any possible remaining linearization row, and not just the following.

Furthermore, the different prediction strategies (top-down and bottom-up), become filterings of the algorithms, since they reduce the number of parse items.

7.3 Implementing and testing the algorithms

The algorithms presented in this paper have been implemented in the programming language Haskell, for inclusion in the Grammatical Framework system [7]. These implementations are described by Burden [2]. We have also started to implement a selection of the algorithms in the programming language Prolog.

Preliminary results suggest that the Active algorithm with bottom-up prediction is a good candidate for parsing grammars written in the Grammatical Framework. For a normal sentence in the English resource grammar the speedup is about 20 times when compared to context-free parsing and filtering of the parse trees. In the future we plan to test the different algorithms more extensively.

Acknowledgments

The authors are supported by the EU project TALK (Talk and Look, Tools for Ambient Linguistic Knowledge), IST-507802.

Bibliography

[1] Pierre Boullier. Range concatenation grammars. In *6th International Workshop on Parsing Technologies*, pages 53–64, Trento, Italy, 2000.

[2] Håkan Burden. Implementations of parsing algorithms for linear multiple context-free grammars. Master's thesis, Göteborg University, Gothenburg, Sweden, 2005.

[3] Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCLR-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[4] Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University and Chalmers University of Technology, November 2004.

[5] Peter Ljunglöf. Grammatical Framework and Multiple Context-Free Grammars. In *9th Conference on Formal Grammar*, Nancy, France, 2004.

[6] Ryuichi Nakanishi, Keita Takada, and Hiroyuki Seki. An efficient recognition algorithm for multiple context-free languages. In *MOL5: 5th Meeting on the Mathematics of Language*, pages 119–123, Saarbrücken, Germany, 1997.

[7] Aarne Ranta. Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[8] Hiroyuki Seki, Takashi Matsumara, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

[9] Stuart Shieber, Yves Schabes, and Fernando Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

[10] Klaas Sikkel and Anton Nijholt. Parsing of context-free languages. In G. Rozenberg and A. Salomaa, editors, *The Handbook of Formal Languages*, volume II, pages 61–100. Springer-Verlag, Berlin, 1997.

[11] K. Vijay-Shanker, David Weir, and Aravind Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Meeting of the Association for Computational Linguistics*, 1987.

[12] Mats Wirén. *Studies in Incremental Natural-Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden, 1992.

[13] Daniel H Younger. Recognition of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

Paper 2

Natural Language Generation from Class Diagrams

Reprint from the proceedings of:

MoDeVVa 2011
MoDELS Workshop on Model-Driven Engineering, Verification and Validation

Wellington, New Zealand
October 2011

Natural Language Generation from Class Diagrams

Håkan Burden[1] and Rogardt Heldal[1]

[1] Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Göteborg, Sweden
{burden, 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.

1 Introduction

In Model-Driven Architecture (MDA; [15, 24]) 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].

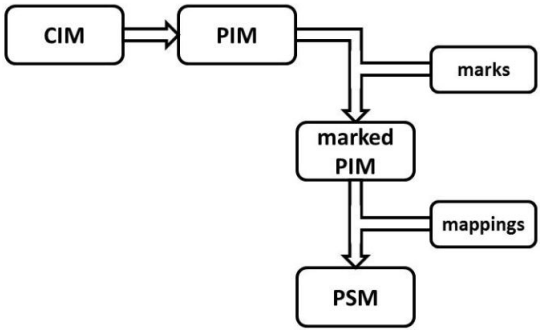

Figure 1: One realisation of the MDA process

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, 32] to convey the intentions and motivations in the CIM as well as correctly describe the PSM [25].

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 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 [19].

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 [19]? 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 [28]. This was achieved by first transforming the marked PIM into a linguistic model defined by the Grammatical Framework [27]. 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, 26, 33]) evolved from merging the Shlaer-Mellor method [29] with the Unified Modeling Language (UML, [22]).

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 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 [22] 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 [30].

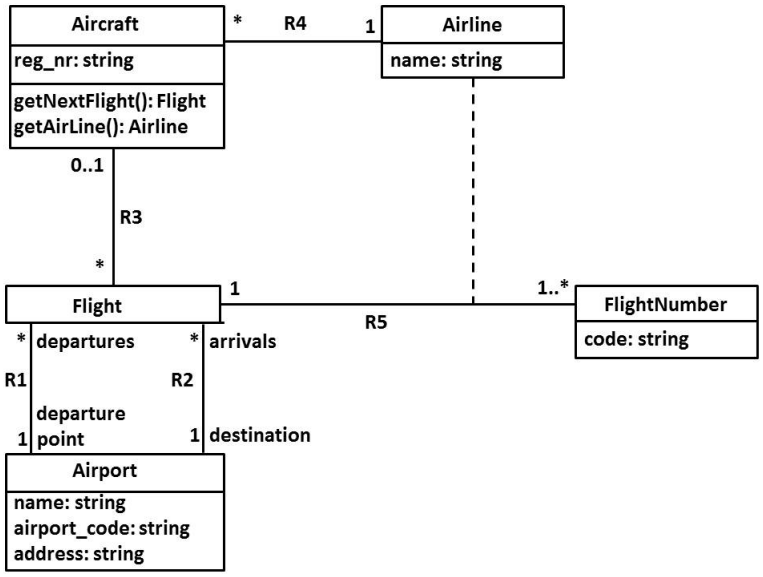

Figure 2: Our example class diagram

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 [28]. 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 [28]. 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, [27]). 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 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.

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

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, $\mathtt{t}$, by a concrete syntax, $\mathtt{C}$, 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

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

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*

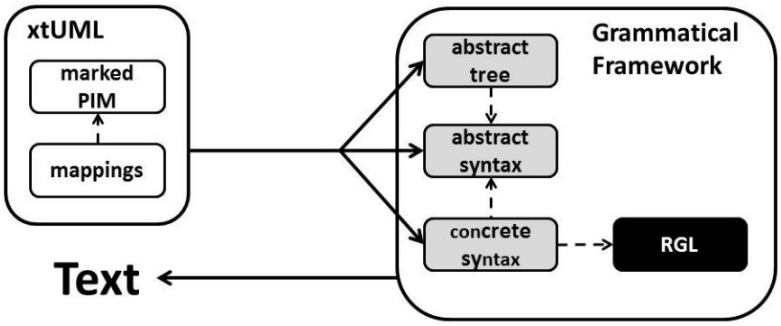

Figure 4: From marked PIM to text

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 transformation. 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, 30]. GF is currently used in collaboration with industry for multilingual translation in the MOLTO-project [18] and has previously been used for multi-modal dialogue systems [4, 34] and in collaboration with the car industry [12].

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; [32] 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 [20]. Of the 24 contributions only one concerned the generation of natural

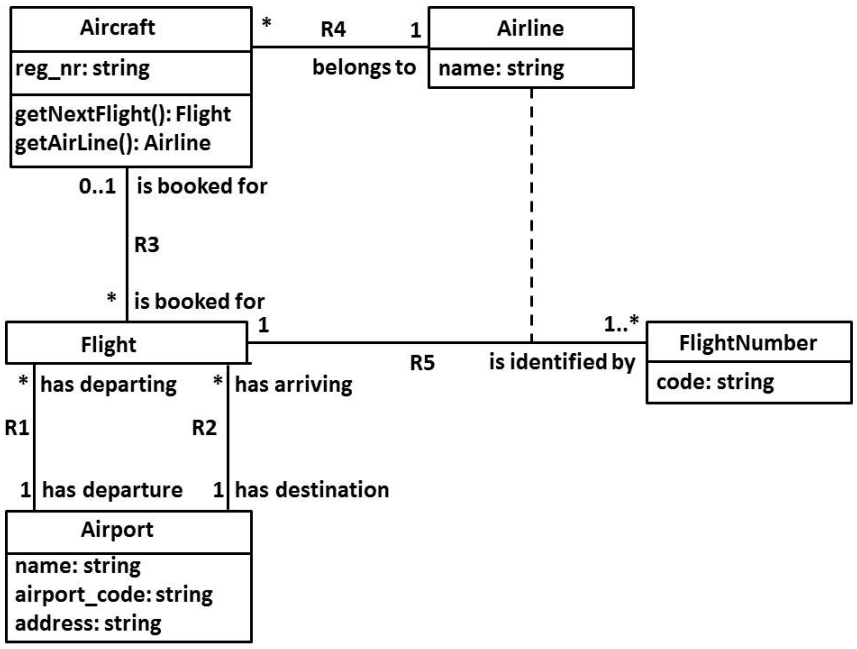

Figure 5: Our class diagram revisited

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. [20] 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, [23]) 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 predefined 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; [21, 35]). 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 [31] 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.

Bibliography

[1] Staffan Andersson and Toni 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] Jim Arlow, Wolfgang Emmerich, and John 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/products/. Accessed 13th January 2012.

[4] Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne 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] Petra Brosch and Andrea 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] David A. Burke and Kristofer Johannisson. Translating formal software specifications to natural language. In Philippe Blache, Edward P. Stabler, Joan Busquets, and Richard Moot, editors, *LACL*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2005.

[7] Jordi Cabot, Raquel Pau, and Ruth 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] Donald Firesmith. Modern requirements specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[10] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2002.

[11] Christian F. J. Lange, Bart Du Bois, Michel R. V. Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[12] Staffan Larsson and Jessica 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] Peter 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] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[16] Farid Meziane, Nikos Athanasakis, and Sophia 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] Molto - Multilingual On-line Translation. http://www.molto-project.eu/. Accessed 1st July 2011.

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

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

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

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

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

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

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

[26] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

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

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

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

[30] Toni Siljamäki and Staffan Andersson. Performance benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[31] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori 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.

[32] Leon 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.

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

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

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

Paper 3

Executable and Translatable UML – How Difficult Can it Be?

Reprint from the proceedings of:

APSEC 2011
18th Asia-Pacific Software Engineering Conference

Ho Chi Minh City, Vietnam
December 2011

Executable and Translatable UML – How Difficult Can it Be?

Håkan Burden[1], Rogardt Heldal[1], and Toni Siljamäki[2]

[1] Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Göteborg, Sweden
{burden, heldal}@cs.chalmers.se

[2] Ericsson AB, Stockholm, Sweden
toni.siljamaki@ericsson.com

Abstract

Executable and Translatable UML enables Model-Driven Architecture by specifying Platform-Independent Models that can be automatically transformed into Platform-Specific Models through model compilation. Previous research shows that the transformations result in both efficient code and consistency between the models.

However, there are neither results for the effort of introducing the technology in a new context nor on the level of expertise needed for designing the Platform-Independent Models. We wanted to know if teams of novice software developers could design Executable and Translatable UML models without prior experiences of software modelling.

As part of a new university course we conducted an exploratory case study with two data collections over two years. Bachelor students were given the task to design a hotel reservation system and the necessary test cases for verifying the functionality and structure of the models within 300 hours, using Executable and Translatable UML.

In total, 43 out of 50 teams succeeded in delivering verified and consistent models within the time frame. During the second data collection the students were given limited tool training. This gave a raise in the quality of the models.

Due to the executable feature of the models the students were given constant feedback on their design until the models behaved as expected, with the required level of detail and structure. Our results show that using Executable and Translatable UML does not require more expertise than a bachelor program in computer science. All in all, Executable and Translatable UML could play an important role in future software development.

1 Introduction

In Model-Driven Architecture (MDA; [24]), the requirements and responsibilities of the system are given a structure by the use of software models in a Computationally-Independent Model, the CIM, often referred to as the domain model [21]. Features such as specific algorithms and system architecture are defined by the next layer of models, the Platform-Independent Models, the PIM. The PIM has no ties towards the hardware nor the programming languages that will in the end realise the system. Such information is added to the Platform-Specific Model, the PSM. As a result the software models of the CIM and the PIM can describe many different implementations of the same system. The models become reusable assets [19] serve both as a description of the problem domain and a specification for the implementation, bridging the gap between problem and solution.

Executable and Translatable UML (xtUML; [32, 18]) is an extension of UML [23] with models that can be executed and translated into code through model compilers. In MDA terms, the xtUML model is an executable PIM that can be automatically transformed into a PSM. The efficient and consistent transformation from a PIM specified using xtUML to a PSM has been tested and proven in previous work [30, 5]. But it is still an open question how much expertise that is required to use xtUML as an executable modelling language for PIMs.

1.1 Motivation

For ten years we have given a university course where teams of students go through the different tasks of an MDA process; from analysis to implementation by designing the system using UML models. The process is illustrated in Figure1. The numbers for each activity in the process are specific to the course and state the maximum number of hours for each student.

The analysis phase was used to capture the business rules of the problem domain in models that satisfy the requirements of the system. The focus during the analysis phase was thus on understanding the problem domain by using activity diagrams, use cases and conceptual class diagrams [23, 14].

The second phase of the process, the design phase, was where more detailed UML diagrams were used such as interaction, state chart, class and component diagrams. Even though this phase should be important for the overall process, we found that it contributed little to the overall system for most teams (in Lean terms the models represent waste [26]). The diagrams were incomplete, lacking necessary details in structure and behaviour. The only way of testing the models was through model inspection, making it a matter of opinion when the models are complete [27]. Another problem with the models was that they were inconsistent with each other leading to complications about which model to follow in the transformation to source code. The impact of the problems vary depending on how important the models are in the development process and when and how the inconsistencies are shown [13, 15, 11, 33].

The design phase was followed by an implementation phase were the students manually transformed their models into Java. This meant that it took months before the students could test their analysis and design. This is a problem shared with industry [17].

Since the code is manually written with the models as a guide it also means that there is a difference in the interpretation of the problem between the UML model and the hand-written code. By default you reanalyze the problem when you start writing the code, and you often come up with a different solution compared to the modelled solution. Eventually the model and the hand-written code diverge, so the only way to really understand what's going on in the system is to study the hand-written code. The model may then serve as a quick, introductory overview of the system, but it may also be incorrect as soon as you stop updating the model for reflecting the changes made to the hand-written code. This notion of architecture erosion is a well-known problem and is still being reported on [25, 16, 3].

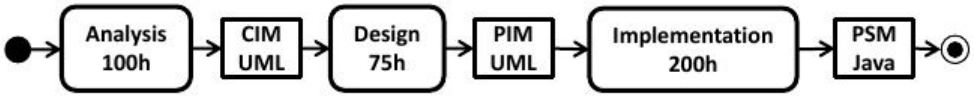

Figure 1: The old software development process

1.2 Aim and Research Question

As a result from collaboration between industry and academia we came up with the idea to use xtUML to model the component- and class diagrams and the statemachines instead of UML in the design phase. If the change of modelling language is successful we will get rid of the inconsistency problems. With an executable PIM it should be possible for the students to test and validate their design decisions without having to implement them in Java first. And when the models behave as expected and the design phase is complete, all functionality of the system should be captured in the models [32, 27]. In the long run this will mean that a lot of the work that was done in the implementation phase can be replaced by generating the code straight from the models, leading to shortened implementation times and consistency between models and code.

Swapping UML for xtUML is not a one-to-one substitution. If it is a matter of opinion when a UML model is complete, an xtUML model is complete when all test cases return the expected results [27]. So, xtUML is more than the graphical syntax, the models have to be given semantics to be executable. In addition, test cases have to be modelled, executed and evaluated. These additions demand that the xtUML tool is more than a drawing tool.

Will the immediate and constant feedback that is given from executing the test cases compensate for the increase in modelling effort? Or will the added effort for learning xtUML take so much time that there is no left for modelling? This concern is re-phrased into our research question:

"Can teams of four novice software modellers solve a problem that is complex enough to require the full potential of xtUML as a modelling language within a total of 300 hours?"

To answer our research question we scrapped our old course in favour of a new one that follows the process seen in Figure 2. Instead of spending 200 hours implementing the design to be able to test and verify it, testing will now be a part of the design phase. Just as for the process in Figure 1 the number of hours in each step states the maximum for each student. The introduction of the new design phase was done as a case study with the ambition to explore and explain the transition and its implications.

1.3 Contribution

Earlier contributions has shown how Executable and Translatable UML enables MDA [18, 27, 19], the reusability of the PIM has been reported on in [2] and the efficiency of the transformations from PIM to PSM is illustrated by [30]. Our contribution shows

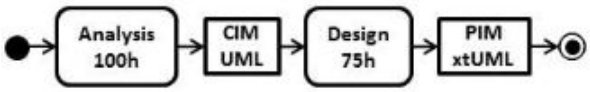

Figure 2: The proposed software development process

that xtUML as a technology is mature enough to to be used by novices to design executable PIMs.

1.4 Overview

In the next section we go into more detail of xtUML and how it can be used. In section III we explain how our subjects made use of xtUML to develop and test a hotel reservation system. Our findings and their validity are presented in section IV. In section V we discuss the implications of our results and in section VI we relate our own case study to previous work. This is followed by a conclusion and some ideas about further investigations regarding the usage of xtUML.

2 Executable and Translatable UML

The Executable and Translatable Unified Modeling Language (xtUML; [18, 32, 27]) evolved from merging the Shlaer-Mellor method [29] with the Unified Modeling Language (UML, [23]).

2.1 The Structure of xtUML

Three kinds of diagrams are used for the graphical modeling together with a textual action language. The diagrams are component diagrams, class diagrams and state-machines. There is a clear hierarchical structure between the different diagrams; state-machines are only found within classes, classes are only found within components. The different diagrams will be further explained below, together with fragmentary examples taken from the problem domain given to the students, a hotel reservation system.

2.1.1 Component Diagrams

The xtUML component diagram follows the definition given by UML. An example of a component diagram can be found in Figure 3. In this diagram the hotel domain depends on the bank for checking that a transaction has gone through as part of the process of making reservations. The User component represents a users of the system and this is where the test cases are placed.

2.1.2 Class Diagrams

In Figure 4 we have an example of an xtUML class diagram. It describes how some of the classes found in the Hotel component relate to each other. I.e. a Room can be

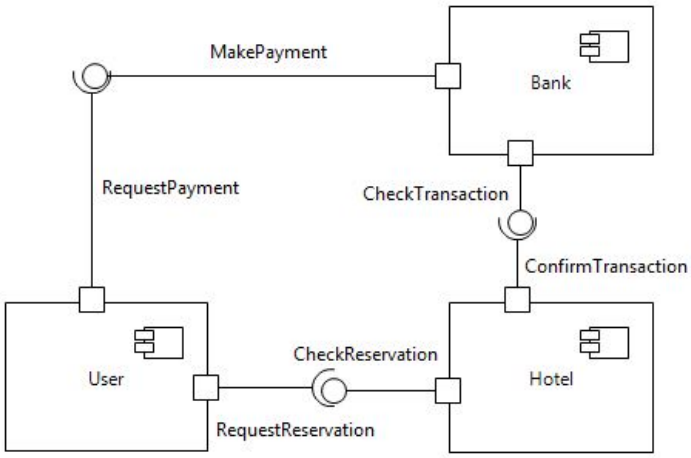

Figure 3: An xtUML component diagram

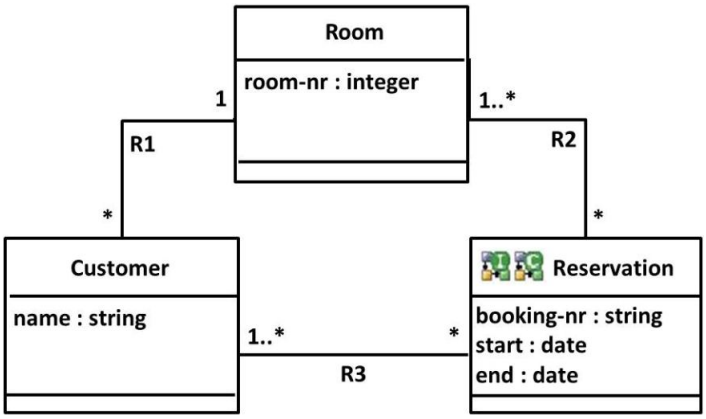

Figure 4: An xtUML class-diagram

related to any number of Reservations (shown by an asterisk, *) but a Reservation has to be related to at least one Room (visualized by 1..*).

The xtUML classes and associations are more restricted than in UML. We will only mention those differences that are interesting for our case study. A feature such as visibility constraints on operations and attributes does not exist. They are therefore accessible from anywhere within the same component. In UML the associations between classes can be given a descriptive association name while in xtUML the association names are automatically given names on the form RN where N is a unique natural number, e.g. Room is associated to Reservation over the association R2.

2.1.3 State Machines

In the class diagram in Figure 4 the Reservation class has both an instance and a class state machine which is indicated by the small figure in the top-left corner of the class. The instance state machine can be found in Figure 5. This state machine covers the first four states of the Reservation procedure, e.g. from the second state, Get rooms, it is possible to reach the third state, Lock rooms, by requesting the rooms. If there are no available rooms you return to the initial state were you can start a new search. Each instance of Reservation has its own instance statemachine that starts running when the Reservation is created.

A class-based state-machine is shared among all instances of a class and starts running as soon as the system starts, like a static process. For shared resources, such as rooms, a class state-machine can be used to ensure that only one reservation instance can book a room at any time.

2.1.4 Action Language

An important difference between standard UML and xtUML is that the latter has a textual programming language that is integrated with the graphical models, sharing the same meta-model [29, 6].

The number of syntactical constructs is deliberately kept small. The reason is that each construction in the Action Language shall be easy to translate to any programming language (such as Java, C or Erlang) enabling the PIM to be reused for different PSMs [2].

There are certain places in the models were Action Language can be inserted, such as in operations, events and states. Over the years a number of different Action Language have been implemented [18] and in 2010 OMG released there own standard [22].

2.2 Interpretation and Code Generation

Since xtUML models have unambiguous semantics all validation can be performed straight on the xtUML model by an interpreter. During the execution of the test cases an object model is created. The object model includes all class instances with their current attribute values and by which associations they are linked to each other. During execution all changes of the association instances, attribute values and class instance are shown [14] as well as the change of state for classes with statemachines in the object model.

The xtUML models can be translated into Platform-Specific Models by model compilers. Since the Platform-Specific code is generated from the model, it is possible for the code and the 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 code. The efficiency of the generated code has been reported on by [30]. [5] have used the model compiler to generate test cases for the PSM.

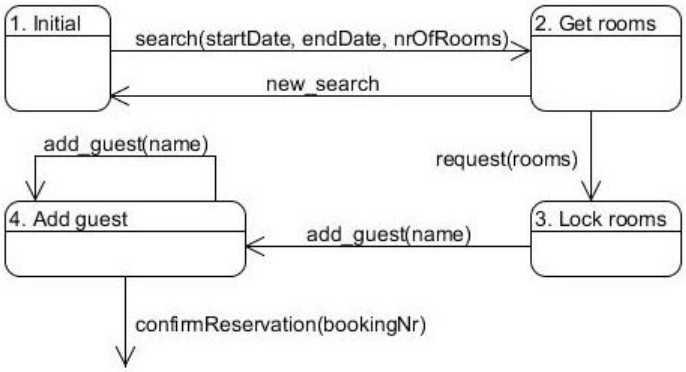

Figure 5: A partial xtUML statemachine

3 Case Study Design

To answer our research question:

"Can teams of four novice software modellers solve a problem that is complex enough to require the full potential of xtUML as a modelling language within a total of 300 hours?"

we have both in 2009 and 2010 let our students use xtUML to design hotel reservation systems. The resulting models have been inspected and compared against our evaluation criteria.

3.1 Subject and Case Selection

3.1.1 The Subjects

Our subjects were students in the final year of their bachelor programs in computer science and software engineering. Their prior knowledge of modelling is limited to class diagrams but they are used to programming in an object-oriented paradigm using Java. In our curriculum the students do two courses in parallel with a working week of 50 hours, so we expect the subjects to work 25 hours a week on our course. A team of four subjects is expected to do a total of 100 hours per week.

3.1.2 The Case

We chose a domain that the subjects could relate to and have some prior knowledge about. The idea is that the subjects shall focus on modelling, not learning a new subject matter. The domain should also have distinct concepts so that an object-oriented solution made sense and have problems where it is natural to use state machines. We also wanted the domain to include problems with algorithmic complexity. Our last requirement was that the domain should represent an open-ended problem so that there is not one right solution. A system for handling hotel reservations seemed to fit all our requirements.

In the hotel domain reservations, customers and rooms are all examples of distinct concepts. The booking process itself has a chain of states that it is natural to control with a statechart, while finding all the possible matches to a set of search criterias for a reservation is an algorithmic problem. These two together, controlling the order of events and searching for rooms, meant that the domain poses the problem of access and allocation of shared and limited resources.

The new design phase was given three weeks, just as the previous design phase. The work was done in teams of four subjects, with a total workload of 75 hours per subject.

The subjects used BridgePoint [4] from Mentor Graphics [20] to design the xtUML models. There were three 90-minute lectures related to xtUML and BridgePoint. Two of these lectures were given by industry representatives, one from the tool vendor Mentor Graphics and one from Ericsson AB as users of BridgePoint. The subjects were encouraged to study xtUML on their own and we recommended them to read [18] and [31]. Each team had half an hour a week with a researcher to discuss design issues. Besides lecturing and supervising on design issues the researchers played the role of project owners.

In 2010 we added limited tool support for the subjects. A subject from 2009 used a total of 22 hours spread over the three weeks to help the subjects of 2010 with BridgePoint. This meant that each team had access to less than an hour of tool training for the entire design phase.

3.2 Data Collection Procedures

We have done two data collections, in 2009 and 2010 respectively. In 2009 there were 88 subjects split into 22 teams. In 2010 we had 108 subjects divided into 28 teams, with four (sometimes three) members per team. We used three forms of data collection; model evaluations, informal discussions and a questionnaire. Model evaluations and informal discussions were used both times but the questionnaire was only used in 2010.

3.2.1 Evaluating the xtUML Models

The evaluation of the xtUML models was done immediately after the design phase and took a whole week to complete. Each team was given 20 minutes to demonstrate their system and to run their tests. Thereafter there was 20 minutes to discuss issues related to their models. Every model was evaluated by two researchers against the evaluation criteria that are specified below. The subjects of each team were present throughout the evaluation, permitting a discussion on the how the criteria on functionality and structure had been interpreted and implemented in the model. A short description of each model with our comments was taken down in a spread sheet.

3.2.2 Informal Discussions

We thought it vital to have an informal opportunity for the subjects to discuss their experiences throughout the design phase. This was an important opportunity for us to get more in-depth information into the problems and discoveries that the subjects had

encountered when using BridgePoint to model xtUML. Since we did not know what to expect for outcome in 2009 we wanted the subjects to have the opportunity to drop us an e-mail, come by our offices or use the lectures for addressing those issues they found urgent. This proved to be a valuable source for data collection, so valuable that we kept it in 2010. The drawback is that it is not a procedure that is always possible to document or systemize.

3.2.3 Questionnaire

One of the most important things that became evident from the informal discussions in 2009 was that the subjects found the learning threshold stressing under the time constraint. Besides introducing tool support in 2010 to ease the subjects' stress we conducted a questionnaire. The aim of the questionnaire was to get a better view of how much time the subjects spent on getting confident in using BridgePoint.

3.3 Evaluation Criteria

Before the subjects started to develop their models we gave them evaluation criteria. The reason was to have a clear idea for both the researchers and subjects of what we expected from the teams.

Based on the use cases from the analysis phase the subjects should come up with executable tests. By running the tests it should be possible to validate that the system is behaving as specified by the CIM. This meant that the object model had to show all relevant changes for objects, associations, attributes [14] and states after a test had been run. At least one test case should be in conflict with the business rules of the system.

However, this does not guarantee that the models are well-structured nor readable [31, 12, 27]. Therefore the criteria enforced an object-oriented design.

For the class diagram we did not accept models with a central object representing the system [31]. Due to the lack of visibility constraints in xtUML we stated that the only way to obtain or change the value of an attribute should be through operations. It should only be possible for a class instance to call another class instance if they are linked by associations. This is so that the dependencies between the class instances are explicit in the class diagram. We wanted all the meaningful associations and concepts in the class diagram. We requested names on classes, attributes, operations and variables that were relevant for the domain.

For the state-machines it was necessary to include all the states and transitions relevant for capturing the lifecycle of the class where it resides. The name of events and states should be meaningful. To ensure that the subjects made use of the power of state machines we required that they should be used for modelling the reservation process.

4 Results

Over the two years that we have used the new design phase we have evaluated 50 xtUML models. Of these 43 have fulfilled the success criteria. The subjects had to

overcome a learning threshold before they got confident in using BridgePoint to develop their xtUML models. All in all, we can answer our research question by stating that the teams did manage to use the full power of xtUML within 300 hours.

4.1 Results from Evaluating the Models

In 2009 all 22 teams came up with an executable model, capturing at least the minimum functionality. 18 of the 22 teams, equivalent to 83% managed to come up with a model within the time frame that met our criteria for a successful model. The models that did not meet the criteria had either a monolithic class that represented the whole system and/or not used the associations to access class instances. Most teams did not use components, but that was not a strict criteria either. This was a shortcoming of the criteria as we had wanted to see the subjects use components, since it would have added a whole new level of abstraction to their models. On the other hand, we were encouraged by the fact that all teams had delivered testable models that fulfilled the criteria for functionality.

Three teams came up with models that went beyond our expectations. They had used components and modelled more functionality than we thought possible. One of the three teams had even made extensive use of design patterns [9].

In 2010 there were 28 teams. 25 of these managed to deliver executable models within the time frame. This is an increase from 82% to 89%, compared to 2009. One of the teams that failed to specify an executable model did so due to unresolvable personal issues within the team. The other two teams misjudged how long time it would take to come up with a model and were not finished in time due to their late start. In 2009 all teams succeeded to come up with executable models compared to 89% in 2010. But this time all teams had components and interfaces with the consequence of using the full power of xtUML. This was not an intended outcome of the added tool training but highly appreciated even so!

4.2 Outcomes From the Informal Discussions

The subjects are used to programming in Java. In 2009 it took the subjects some time before they started to reason about objects and programming in an xtUML-way. When they encountered a problem many of them expected a solution using detailed Java-specific datastrucutres or libraries. In contrast BridgePoint is mostly used for embedded systems were the companies have their own private libraries.

Particularly state-machines were difficult to use since they had no counterpart in Java. It is not possible for us to say if this is due to that Java is the only language they are used to or if it is due to the more abstract level of reasoning in xtUML.

BridgePoint is a powerful tool; enabling modelling, execution of models and translation to source code. All the functionality makes it a complex tool. BridgePoint is also to a large extent menu-driven — many of the design choices are implemented by choosing from drop-down menus and tool panels. The challenge is to get used to all the different combinations of choices that are needed for elaborating the design. The version we used is a plug-in for Eclipse which in itself has a number of features to get used to.

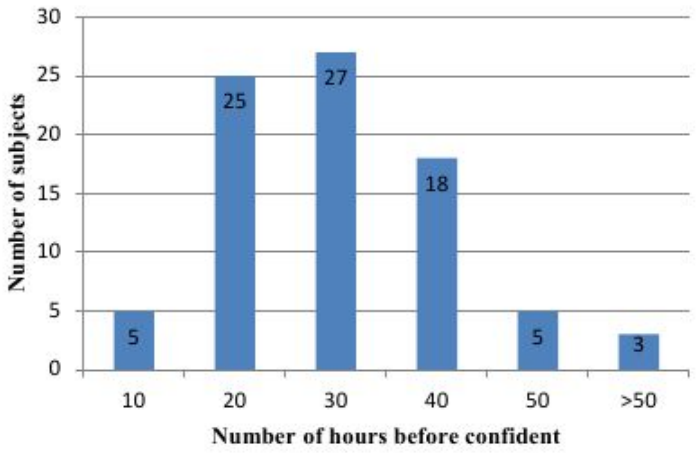

Figure 6: Number of hours that the subjects needed to become confident in using BridgePoint

In reaction to the problems concerned with BridgePoint as a tool we decided to use one of the subjects from 2009 for tool training in 2010. The intention was that this would mean less time spent on understanding the tool and more time to spend on developing the models.

In 2010 the subjects requested a version control system so that they could more easily split the design work between themselves. This was never an issue in 2009 and a sign of more confident subjects. If this is a consequence of the tool training or not is to early to answer.

4.3 Experienced Learning Threshold

In 2010 we used a questionnaire to get a better idea of how the subjects experienced BridgePoint. The question we asked was "How many hours did you spend learning BridgePoint before you got confident in using the tool?"

In total 90 of 108 subjects answered the question. Besides the answers given in Figure 6 six subjects answered that they never became confident and one subject had no comment. The number of hours it took to become confident are given on the x-axis, the number of subjects for a given number of hours is displayed along the y-axis. This means that 27 subjects answered that it took 30 hours to become confident in using BridgePoint.

In Figure 7 the x-axis carries the same information as in Figure 6 while the y-axis now displays the total number of confident subjects for a given time, e.g. after 30 hours a total of 57 subjects felt confident in using BridgePoint. After 40 hours this figure had risen to 75 subjects.

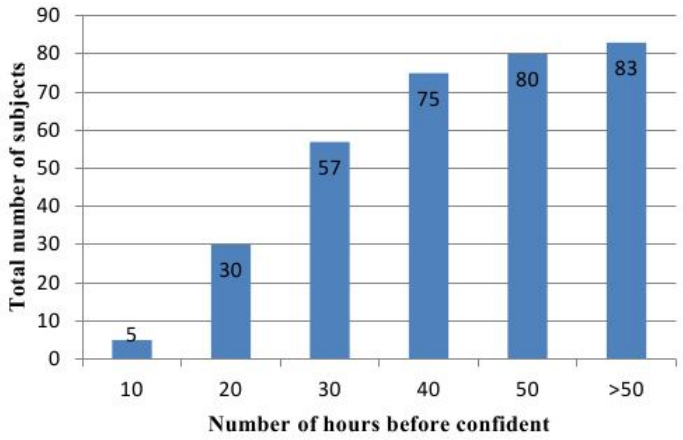

Figure 7: Total number of subjects that are confident in using BridgePoint depending on time

4.4 Relevance to Industry

We wanted to solve our consistency problems by using xtUML instead of UML. In our view this was successful. But we believe that xtUML can have a larger impact than just solving the problems we had with our old development process 1. From the outcomes of the the evaluation of the models, the subjects' own figures for the confidence threshold for using xtUML and the informal discussions we propose a hypothesis.

We state that xtUML, both as a modelling language and tool, is easy to learn and use without any prior experience of software modelling. It is enough to have the programming experience equivalent to a bachelor student in computer science. Our subjects managed to learn and understand the full expressivity of xtUML within 75 hours, and that includes using asynchronous events in statemachines and sending and receiving signals between components as well as designing the models to an appropriate level of detail. If it is possible for us to manage with the transition from UML to xtUML it should be possible to do so in industry as well.

4.5 Evaluation of Validity

We have analysed our results using the classification of validity as defined by [34, 35] and [28].

4.5.1 Construct Validity

Our solution to the problems we had earlier is set in the same context as the problem was. Our subjects have the same background and experience as previous students, just as before the subjects have the necessary domain knowledge from the analysis

phase, they work under the same time constraints and come up with the same kind of models. The only thing that has changed is the modelling language, which is what we want to evaluate.

The subjects were present during the model evaluation. This was done in order to reduce our bias in interpreting their work and how it related to our validity criteria.

In order to assess the quality of the software models we have specified evaluation criteria. In 2009 only three of 22 teams defined two or more components and the interfaces between them. Since the 28 teams in 2010 also had models with several components with defined interfaces we can see that the subjects managed to use the full power of xtUML.

The figures for the number of hours to overcome the learning threshold of BridgePoint were estimations done by the subjects themselves. There might be variations among the subjects of the definition of when the threshold is passed. Our experiences from Ericsson and our own observations correlate with the estimations of the subjects. However, there is a possibility that some subjects have exaggerated or under-estimated their figures due to social factors (reactions towards the tool, researchers or team members). The exact nature of the learning threshold for xtUML will be dependent on the background of the subjects and which tool that is used.

4.5.2 Internal Validity

The evaluated models are developed on the basis of the CIM from the analysis phase, so that everything the subjects need to know about hotel reservation systems should be found in their CIM. This means that they should not need to spend time during the design phase on anything else than learning and using xtUML.

Even if the evaluation criteria have influenced the nature of the results, by defining what we expected from the subjects' models, the process of getting there was by using xtUML.

In the first run of the experiment we did not know what to expect for results. In the second run we had expectations based on the results from the first run. Therefore we were careful to make sure that we as researchers had the same roles towards the subjects in both runs, which led us to let a subject from 2009 take care of the tool training in 2010. We still cannot neglect that our changed expectations might have influenced the outcome in 2010, even if we did not get the results we were expecting. On the other hand this is always the case in situations where you want to replicate research that involves human beings. This is also a threat to the reliability of our results.

It is possible that the subjects have been sharing insights and experiences throughout the case study. We knew this could be the case from the start and that was one reason why we wanted an open-ended problem. During the evaluations we have seen 50 unique models which implies that all teams have had their own process to come up with the models.

4.5.3 External Validity

Today's students are tomorrow's employees. If our subjects can master xtUML it should also be possible for software developers in industry to do so. This is also claimed by [10] who state that students can be used in research as long as it is the evaluation of the use of a technique by novice users that is intended. We can expect software developers in industry to have at least the same competence as our subjects.

The size of the problem given to the subjects is smaller than most indutrial sized tasks. Our domain has a certain level of complexity and was chosen from our collaboration between industry and academia. By handling the access and allocation of the shared resources within the hotel domain, we made sure that the subjects had to solve a non-trivial task.

4.5.4 Reliability

Since the evaluation of the models is subjective there was at least two researchers present at every evaluation in order to reduce the risk of bias and inconsistencies between evaluators. We also used the criteria for a successful model to ensure that the evaluation is less subjective, making the results less dependant on a specific researcher.

BridgePoint was chosen by the authors based on the fact that a team within Ericsson think that this is one of the best MDA solutions today. After we had made our decision on which tool we would prefer to use we contacted Mentor Graphics in order to start a collaboration with them. Mentor Graphics never influenced us on which tool to use. Using another xtUML tool might give other results, especially regarding the threshold, both in figures and what is seen as problematic.

5 Discussion

In our previous MDA process, given in Figure 1, our students manually transformed the PIM into a PSM. From our previous experiences of using xtUML as a code generator [30, 5] we know that this manual transformation can be automated. However, to raise the quality of the generated code the PIM needs to be manually enhanced by a marking model [21, 19, 30]. The generated code will then be sufficient for an embedded system. For systems that interact with human users it will also be necessary to develop the needed user interfaces. All in all the introduction of xtUML should enable a less time-consuming implementation phase compared to our old MDA process.

6 Related Work

There is a previous experience from using xtUML in the context of computing education reported in [8, 7]. One of their motivations for using xtUML in a modelling course is that they found UML to large, ambiguous and complex. In contrast, xtUML models are unambiguous and easier to understand than UML models. The possibility of verifying the models to see if they meet the requirements is important in order to

give the students feedback on their modelling. The authors have used xtUML both for specifying a web application and for 3D drawing software.

By using xtUML in a similar context as ours their work strengthens our claim that xtUML can be used by novice software modellers. However, there work does not report any results from letting undergraduate students use xtUML; there are no clear criteria for what was seen as a successful project and subsequently no reports on how many students that managed to complete the task. It is also unclear how much time the students spend on their models. Another important difference is that we find UML useful for defining the CIM.

Both [18] and [27] describe xtUML in the context of MDA. Starting of from use cases they develop PIMs by using xtUML. While the main focus is on developing an executable PIM both books takes the reader from CIM to PSM. The main differences lie in the choice of tool and in how they choose to describe and explain xtUML.

We have made extensive use of both books as a source of inspiration for how to work with xtUML and as recommended literature for the subjects for obtaining executable PIMs in an MDA context. These are the most cited books on how to use xtUML and they are detailed in how this is accomplished. However, they do not mention the effort for learning xtUML nor the level of expertise needed to use xtUML as a modelling language for PIMs.

7 Conclusions and Future Work

7.1 Summary

Even if the subjects spend the first week of the design phase in order to learn BridgePoint the fact that the models have a testable behaviour more than compensates for the increase in effort. Our subjects used the test cases to refine their models until they met the criteria. As a consequence their PIMs had the necessary detail and structure as defined by the CIM. This was possible since xtUML gave them constant and immediate feedback on all their design decisions.

In contrast, UML models are not executable. More or less the only way of checking the quality of a UML model is by performing a model review. This is a powerful method for improving on UML models but it is also time consuming. And it can be hard to catch the mistakes in complex systems. It becomes a matter of opinion when a model can be considered complete. In contrast an xtUML model is complete when it only delivers expected output for all test cases.

Previous work has shown that xtUML enables MDA by the reusability of the PIMs, the efficient transformation from PIM to PSM and by solving the problems of inconsistencies within the PIM. Our work shows with what little effort and expertise it is possible to develop PIMs, using the full expressivity of xtUML. This implies that Executable and Translatable UML is a technology that is ready to be used within industry.

7.2 Future Work

We are looking at the possibilities to expand the new course so that it covers the entire MDA-process, from CIM to PSM. Issues we want to investigate is how difficult it is to mark the PIMs for an efficient transformation to PSMs, the effort for deploying the generated code on a platform with the required user interfaces and how much time we can save compared to the old development process, illustrated in Figure 1.

In addition, we want to investigate the reasons behind the socio-technical gap [1] to understand why xtUML is not used more within industry and software development.

Acknowledgment

The authors would like to thank Staffan Kjellberg at Mentor Graphics; Stephen Mellor; Leon Starr at Model Integration; Dag Sjøberg at University of Oslo; Jonas Magazinius, Daniel Arvidsson, Robert Feldt and Carl-Magnus Olsson at Computer Science and Engineering in Gothenburg.

Bibliography

[1] Mark S. Ackerman. The intellectual challenge of cscw: The gap between social requirements and technical feasibility. *Human-Computer Interaction*, 15:179–203, 2000.

[2] Staffan Andersson and Toni 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.

[3] Jan Bosch. Architecture in the age of compositionality. In Muhammad Babar and Ian Gorton, editors, *Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin, Heidelberg, 2010.

[4] BridgePoint. http://www.mentor.com/products/. Accessed 13th January 2012.

[5] Federico Ciccozzi, Antonio Cicchetti, Toni Siljamäki, and Jenis Kavadiya. Automating test cases generation: From xtUML system models to QML test models. In *MOMPES: Model-based Methodologies for Pervasive and Embedded Software*, Antwerpen, Belgium, September 2010.

[6] Michelle L. Crane and Jürgen Dingel. Towards a formal account of a foundational subset for executable uml models. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 675–689. Springer, 2008.

[7] S Flint and C Boughton. Executable/Translatable UML and Systems Engineering. In Alan McLucas, editor, *Systems Engineering and Test Evaluation Conference (SETE 2003)*, Canberra, Australia, 2003.

[8] Shayne Flint, Henry Gardner, and Clive Boughton. Executable/Translatable UML in computing education. In *ACE'04: Proceedings of the sixth conference on Australasian computing education*, pages 69–75, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[10] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28:721–734, August 2002.

[11] Christian F. J. Lange. Improving the quality of UML models in practice. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 993–996. ACM, 2006.

[12] Christian F. J. Lange, Bart Du Bois, Michel R. V. Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[13] Christian F. J. Lange and Michel R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 401–411, New York, NY, USA, 2006. ACM.

[14] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[15] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[16] N. Mellegård and M. Staron. Methodology for requirements engineering in model-based projects for reactive automotive software. In *European Conference on Object-oriented Programming (ECOOP)*, Paphos, Cyprus, 2008.

[17] Niklas Mellegård and Miroslaw Staron. Characterizing model usage in embedded software engineering: a case study. In Ian Gorton, Carlos E. Cuesta, and Muhammad Ali Babar, editors, *ECSA Companion Volume*, ACM International Conference Proceeding Series, pages 245–252. ACM, 2010.

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

[19] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[20] Mentor Graphics. http://www.mentor.com/. Accessed 13th January 2012.

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

[22] OMG. Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF). http://www.omg.org/spec/ALF/. Accessed 30th April 2011.

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

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

[26] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[27] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[28] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

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

[30] Toni Siljamäki and Staffan Andersson. Performance benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[31] Leon 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.

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

[33] Ragnhild Van Der Straeten. Description of UML Model Inconsistencies. Technical report, Software Languages Lab, Vrije Universiteit Brussel, 2011.

[34] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers Norwell, MA, USA, 2000.

[35] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, California, fourth edition, 2009.