Translating Platform-Independent Code into Natural Language Texts

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

Keywords:
Model Transformations: Executable and Translatable UML: Grammatical Framework

Abstract:
Understanding software artifacts is not only time-consuming, without the proper training and experience it can be impossible. From a model-driven perspective there are two benefits from translating platform-independent models into natural language texts: First, the non-functional properties of the solution have already been omitted meaning that the translations focus on describing the functional behaviour of the system. Second, the platform-independent models are reusable across platforms and so are the translations generated from them. As a proof-of-concept a platform-independent Action language is translated into natural language texts through the framework of model transformations.

1 INTRODUCTION

In MDA the platform-independent model, PIM, should be a bridge between the specifications in the computationally-independent model, CIM, and the platform-specific model, PSM (Miller and Mukerji, 2003; Mellor et al., 2004). Thus it is important that the PIM is clear and articulate (Lange et al., 2006) to convey the intentions and motivations in the CIM as well as correctly describe the PSM (Perry and Wolf, 1992).

Since the PSM can be automatically generated from the PIM all changes to the software can be done at PIM-level or on the transformations. In this way the PIM and the PSM are in synchronisation with each other. To keep the CIM and the PIM synchronised is not as easy since their are no automatic transformations from CIM to PIM, yet. Here the translation of the PIM into textual representations can serve as a means of validation of the PIM, in regard to the CIM, during development or to make it easier for new developers to comprehend the structure and behaviour of the system (Arlow et al., 1999).

Claims have been made that comprehensibility is more important than completeness if models are used for communication between stakeholders (Mohagheghi and Aagedal, 2007). But if the stakeholders want to know if the PIM is correct with regards to the software specifications, completeness is just as important. Understanding the annotation and testing of a model requires an understanding of object-oriented design, knowledge of the used models and experience of using the modelling tools (Arlow et al., 1999). Natural language on the other hand is suitable for stakeholders without the necessary expertise in models and tools (Spreeuwenberg et al., 2010).

Contributions

This paper shows i) how a platform-independent Action language can be translated into natural language texts ii) by putting natural language generation of software behaviour within the perspective of model-driven software development iii) with transformation rules that are reusable across domains and platforms.

Overview

Section 2 presents the theoretical framework for the study. The tools, technologies and transformations that are used in the study are explained

together with examples of translations in section 3. The study is then put in a more general context in the discussion, section 4, before the conclusion is given in section 5. Finally, possibilities to further explore the results are presented in section 6.

Java statement

```
if (saveAuctions())
```

English translation

/* If save auctions succeeds */

Figure 1: Example translation of Java to English

2 THEORETICAL FRAMEWORK

2.1 Natural Language Generation

Natural Language Generation (NLG; (Reiter and Dale, 1997)) is a theoretical framework for describing the transformation from software internal models of information into natural language representations. 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. Traditionally NLG is broken down into a three-stage pipeline; *text planning*, *sentence planning* and *linguistic realisation* (Reiter and Dale, 1997).

Text Planning Text planning is to decide on what information in the original model to communicate to the readers.

Sentence Planning The second stage defines the structure of the individual sentences. This is also the time for choosing the terms that are going to be used for the different concepts. The original software model has now been transformed into an intermediate linguistic model, a grammar.

Linguistic Realisation In the last stage the linguistic model is used to generate text with correct word order and word forms. Through the linguistic realisation the intermediate model has been transformed into natural language text.

2.2 Related Work

Nicolás and Toval (Nicolás and Álvarez, 2009) provide a systematic literature review on the textual generation from software models. This is a good starting point for a broader investigation into the topic. In their study there is no evidence of text generation from platform-independent Action languages that specify software behaviour.

Recently there has been a flourish of publications on generating natural language from source code. Rastkar et. al. (Rastkar et al., 2011) generate English for crosscutting concerns, functionality that is defined in multiple modules, from Java code. As a result of the scattered nature of the crosscutting concerns they are difficult to handle during software evolution. Having a natural language summary for each part of the concern and where it is implemented helps developers handle software change tasks. Sridhara et. al. (Sridhara et al., 2010; Sridhara et al., 2011) have also investigated natural language generation from Java code. Their motivation is that understanding code is a time consuming activity and accurate descriptions can both summarise the algorithmic behaviour of the code and reduce the amount of code a developer needs to read for comprehension. The automatic generation of summaries from code mean that it is easy to keep descriptions and system synchronized. An example of a translation from Java to English is found in Figure 1, taken from (Sridhara et al., 2010). Another approach to textual summarisations of Java code is given by Haiduc et. al. (Haiduc et al., 2010). They claim that developers spend more time reading and navigating code than actually writing it. Central to these publications is that they have to have some technique for filtering out the non-functional properties from the source code before translation into natural language.

There are also contributions on using grammars to translate platform-independent specifications into natural language. One such attempt is the translation between the Object Control Language (OCL; (Warmer and Kleppe, 2003)) and English (Hähnle et al., 2002; Burke and Johannisson, 2005). This work was followed up by a study on natural language generation of platform-independent contracts on system operations (Heldal and Johannisson, 2006), where the contracts were defined as OCL constraints and specified the pre- and post-conditions of system operations.

3 EXPLORATORY CASE STUDY

In order to explore how a platform-independent Action language can be translated into natural language texts *Executable and Translatable UML* is used to encode the PIM and define the transformation rules. Instead of generating text straight from the PIM the *Grammatical Framework* works as an intermediate modelling language to handle the linguistic properties of the text. In this way the MDA process is integrated with the process of natural language generation.

3.1 Executable and Translatable UML

Executable and Translatable UML (xtUML; (Starr, 2001; Mellor and Balcer, 2002)) evolved from merging the Schlaer-Mellor methodology (Shlaer and Mellor, 1992) with the UML[1] and is a graphical programming language for encoding platform-independent models. BridgePoint[2] was chosen as the xtUML tool.

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. Action language can be used in all three component types to define their functional behaviour. The diagrams and action language will be further explained using simplified examples taken from the problem domain chosen for the proof-of-concept implementation, a hotel reservation system.

3.1.1 Diagrams

The xtUML component diagram follows the definition given by UML. In Fig. 2 there is an example of a component diagram. It consists of two components, Hotel and User, connected across an interface.

Fig. 3 shows the class diagram that resides within the Hotel component in the component diagram. The xtUML classes and associations

[1] http://www.uml.org/

[2] http://www.mentor.com/products/sm/model_development/bridgepoint/

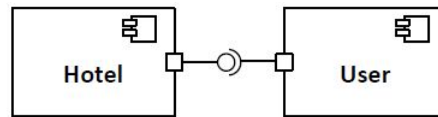

Figure 2: An xtUML component diagram

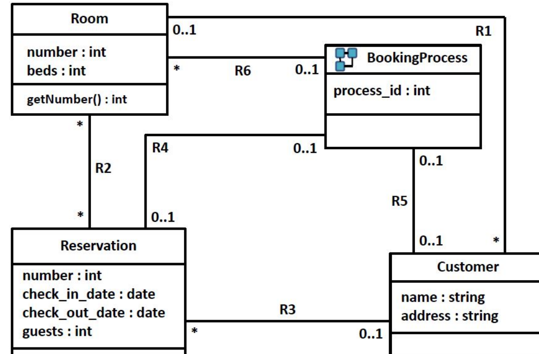

Figure 3: An xtUML class-diagram

are more restricted than in UML. Only those differences that are interesting for the case study are mentioned. 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. In Fig. 3 Room is associated to Reservation over the association R2. The BookingProcess has no operations, instead the dynamic behaviour is defined by the statemachine residing within, marked by the icon in the top-left corner of the BookingProcess class.

In xtUML a statemachine comprises states, events, transitions and procedures (Mellor and Balcer, 2002). Fig. 4 shows the statemachine that describes the lifecycles of individual instances of a BookingProcess. Given the statemachine there are two possible transitions from the state Searching; either the event add_room is triggered and the BookingProcess transits to the Adding rooms state or cancel is triggered and the new state is Canceling. If another event is triggered while a BookingProcess is in the Searching state, the event is either ignored or an error is thrown. The states can contain procedures, both events and procedures are defined by the Action language.

3.1.2 Action Language

An important property of xtUML is the Action language. It is a textual programming language that is integrated with the graphical models, sharing the same meta-model (Shlaer and Mellor, 1992). Since the Action language shares the same

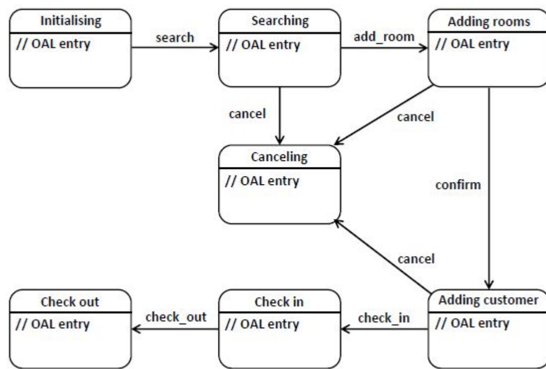

Figure 4: An xtUML statemachine

metamodel as the graphical models it can be used to define how values and class instance are manipulated (Larman, 2004) as well as how the classes change their state (Shlaer and Mellor, 1992). Action language can be used to define the calls between the components as described by the interfaces or to control the flow of calls through the ports of the components. An example of how the Action language can be used is given in Fig. 7. The code details a simple algorithm for finding available rooms and resides within the Searching state of Fig. 4. The example will be further explained in section 3.

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 enabling the PIM to be reused for different PSMs. Over the years a number of different Action languages have been implemented (Mellor and Balcer, 2002) and in 2010 OMG released there own standard, ALF[3].

3.1.3 Translating the Models

The xtUML model can be translated into a Platform-Specific Model by a model compiler. A model compiler traverses the metamodel of the PIM and maps each concept into the corresponding concepts of the target language, while preserving the structure of the PIM. 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.

[3] http://www.omg.org/spec/ALF/

The abstract syntax:

```
cat Exp
fun Sum  : Exp × Exp → Exp
    EInt : Int → Exp
```

The concrete syntax:

```
lincat Exp = Str
lin    Sum n m = "the sum of" ++ n ++
                 "and" ++ m
       EInt  i = i.s
```

Figure 5: A small GF grammar

3.2 Grammatical Framework

Grammatical Framework (GF[4]; (Ranta, 2011)) is a domain-specific language for defining Turing complete grammars (Chomsky, 1959).

3.2.1 GF Grammars

GF separates the grammars into abstract and concrete syntaxes (Mccarthy, 1962). 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 an NLG view the categories are the content and the functions the structure of the text. In the concrete syntax each category and function is given a linearisation definition (`lincat` and `lin` respectively). These definitions gives the sentences their structure and the terminology to be used for the concepts.

A small example of a GF grammar is given in Fig. 5. In the concrete syntax the linearisation of expressions is defined as strings. Integers are represented by their string values which are obtained by record selection, `i.s` (Ranta, 2011). The linearisation rule for `Sum` is then defined by concatenating the string arguments into their corresponding slots.

An abstract syntax tree defines in which order the functions of the abstract syntax are to be used. A text with multiple readings is ambiguous and will return an abstract tree for each possible reading but each tree will only return one text.

Given the example above the sentence *the sum of 3 and 5* will have the tree

```
Sum (EInt 3) (EInt 5)
```

The transformation from abstract tree to text is called linearisation. Linearisation corresponds to the linguistic realisation of NLG. This trans-

[4] http://www.grammaticalframework.org/

formation is a built-in property of GF (Ljunglöf, 2011; Angelov, 2011).

3.2.2 The GF Resource Library

In the Resource Grammar Library (RGL; (Ranta, 2009)) a common abstract syntax has 24 different implementations in form of concrete syntaxes. Among the concrete languages are English, Catalan and Japanese. The resource grammars have a shared interface which hides the complexity of each concrete language behind abstract function calls. Just as a programmer can use a Java API without knowing how the methods are implemented, the resource grammars support grammar development through an interface that specifies how grammatical structures can be developed (Ranta, 2008). The implementation of each function can be retrieved from the source code and its documentation.

3.3 Model-to-Text Transformations

The automatic translation from software models to natural language texts consists of two transformations, see Fig. 6. 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 the linguistic realisation when the linguistic model is used to generate natural language text.

Both transformations are examples of unidirectional and automatic transformations (Stevens, 2007). The first transformation is a reverse engineering translation since the level of abstraction is higher in the target models than in the source models and the two models are defined by different metamodels (Mens and Gorp, 2006).

Each transformation consists of a set of rules (Kleppe et al., 2005) and an algorithm for how to apply the rules (Mellor et al., 2004). Since the rules of both transformations are defined according to their respective meta-models they are reusable for all models that conform to the same meta-model (Atkinson and Kuhne, 2003; Mellor et al., 2004). The transformations can even be applied to partial xtUML models, enabling textual feedback throughout development on all changes and updates, even if the models need further refining.

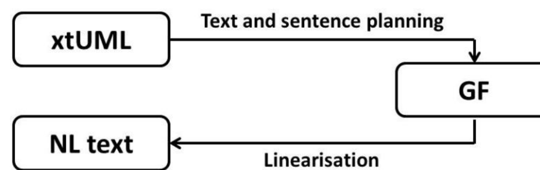

Figure 6: From platform-independent models to natural language texts

3.4 Defining the Grammar

The abstract grammar of the Action language specifies two main categories, expressions and statements. Expressions can be of two kinds, *sentences* or *noun phrases*.

3.4.1 Expressions

From a linguistic point of view a sentence, abbreviated as S, expresses a proposition about the world it inhabits. An example from the Action language is `x == y`, represented in English as *x equals y*. The proposition itself does not claim to be true or false, that is dependent on the context of its evaluation. A characteristic of English propositions are that they follow the form subject-predicate-object, in the example above x is the subject, *equals* is the predicate and y is the object.

In natural languages, both subjects and objects can have more complicated structures, an example being *the sum of n and m*, written `n + m` in Action language. Such a structure is referred to as a noun phrase, abbreviated as NP. The result of combining the two examples is the expression `x == n + m`, translated as *x equals the sum of n and m*. (Expressions such as `x == y == n + m` can not be formed since the expressions on either side of the equality sign have to refer to members of the program. From a linguistic point of view the expressions have to be NPs.)

This distinction between expressions as sentences and noun phrases is captured in the abstract grammar by the two categories `SExpr` and `NPExpr`. The abstract syntax for the equality function then becomes

```
equality : NPExpr × NPExpr → SExpr
```

with the concrete syntax for English defined using the resource grammars

```
equality x y = mkS (pred (mkV2 "equal" x y)
```

The function `mkV2` takes a string value and returns a verb that expects two NPs, a subject (`x`) and an object (`y`). The function `pred` then takes the verb and the two NPs in order to return an

intermediate structure that is passed on to `mkS`. The result of applying `mkS` is a sentence on the form *x equals y* where both *x* and *y* can be complex NPs. In order to handle agreement between subject and verb the linearisation categories for nouns and verbs have to be more complex than just strings. Exactly how complicated is not a problem for those using the RGL as an API for grammar development, it has already been dealt with by the RGL developers. Instead, the complexity lies in applying the appropriate functions from the API in the right order.

Both the S- and NP-expressions are derived from the xtUML metamodel where they are encoded as subtypes of the metaclass Value or as instances of Variable. In the above example for `equality` both the binary operation and the `NPExpr` are defined as Values. By recursively analysing the left and right expressions of the operation shows that `x` and `y` are instances of the metaclass Variable with their respective names. Unary operations, attribute references and parameters for events and operations are other subclasses of Value.

3.4.2 Statements

If expressions could be both noun phrases and sentences, all statements are sentences. An example of this is the Action language's return statement `return x` where `x` could be both an NP such as *the sum of n and m* as well as a sentence, *n equals m*. The solution is to have two abstract functions defining the return statement, one for returning noun phrases and one for returning sentences

```
returnNP : NPExpr → Stmt
returnS  : SExpr  → Stmt
```

For the concrete syntax a more general phrasing than *return n* is used since it can be unclear for non-programmers to whom *n* is returned and what this means. This decision highlights how the abstract syntax defines the text planning of the natural language generation while the concrete syntax defines the words to be used for different concepts and how these words are to be strung together, i.e. the sentence planning.

The first function for return statements is implemented in a fashion similar to the one used for equality expressions

```
returnNP n =
  mkS (pred n (mkNP the_Det (mkN "result")))
```

and returns statements such as *the result is the sum of n and m* for `return n + m`. For returning sentences other functions from the RGL are used since the type of the argument is different

```
returnS  s =
  mkS (mkCl (mkNP the_Det (mkN "result")) s)
```

As an example *the result is x equals y* is the equivalent translation for `return x == y`.

Finally, a program is defined as a list of statements

```
fun sequence : [Stmt] → Prgm
```

3.5 Translations

The diagram in Fig. 7 shows an example of a program written in Action language side-by-side with its translation where the Action code resides within the Searching state shown in Fig. 4. The generated text is an example of a controlled natural language (CNL; (Wyner et al., 2010)) where the described language is a subset of a natural language. A common aspect of such languages is that they are perceived as lacking in naturalness (Clark et al., 2009) and that the sentences have a repetitive structure inherited from the source model. This can also be a benefit since it allows readers to quickly recognise and interpret the different sentence structures (Clark et al., 2009).

The Action language is platform-independent in the sense that it makes no assumptions on how collections are to be implemented, all collections are treated as sets. This is exemplified on line 6 where many `Rooms` are selected and stored as a set using the variable `rooms`. On line 7 a for-loop is used to iterate over the set. On the other side, the Action language is not independent from the object-oriented modelling paradigm. This shows in lines 1 and 2 where an instance of an object is created and then associated to another object. To interpret the Action language requires an understanding of the implicit information encoded in the paradigm of object-oriented languages (Arlow et al., 1999). The aim of the translation is to make such information explicit without being to lengthy. Another aspect of the underlying design choices of the Action language is shown in the naming convention for traversing across associations. Here the unique association names are used, which have no relevance for the domain. In the translation to natural language texts association names, such as `R2`, are therefor not mentioned.

Just as graphical models the Action language is supposed to deliver a high-level view of the system. But the abstraction gets muddled by

```
create object instance res of Reservation;      res refers to a Reservation
relate res to self across R4;                   res and the BookingProcess share information
res.check_in = param.in;                        res's check in gets the value of the given in
res.check_out = param.out;                      res's check out gets the value of the
                                                    given out
res.guests = param.quantity;                    res's guests gets the value of the given
                                                    quantity
room_number = 0;                                room number gets the value of 0
select many rooms from instances of Room;       rooms refers to many Rooms
for each room in rooms                          for each room in rooms
  relate self to room across R6;                the BookingProcess and room share information
  select many ress related by                   ress refers to many Reservations
    room -> Reservation[R2];
  for each res in ress                          for each res in ress
    if (res.check_in > param.out                if res's check in is greater than the
                                                    given out
     or res.check_out < param.in)               or res's check out is less than the given in
     and room.beds == param.quantity            and room's beds equals the given quantity
      room_number = room.getNumber();           then room number gets the value of room's
                                                    get Number
      break;                                    %the for-loop is terminated
    end if;
  end for;
  if room_number > 0                            if room number is greater than 0
    break;                                      %the for-loop is terminated
  end if;
end for;
if room_number == 0                             if room number equals 0, then a cancellation
  send HotelInterfaces::                        with process id and message is sent to User
    cancellation(process_id:self.process_id,
      message:"No available rooms.");
else                                            else a confirm room with process id and room
  send HotelInterfaces::                        is sent to User
    confirm_room(process_id:self.process_id,
      room:room_number);
end if;
```

Figure 7: An example of Action language code with natural language summarisation

language-specific details such as the association names and the object-oriented syntax, concepts that are not meaningful to all stakeholders (Forward and Lethbridge, 2008).

The generated text is dependent on that meaningful values have been assigned to class names, parameters etc. If the class Reservation was named RSV instead the translation would generate sentences such as *res refers to an RSV* making the generated texts harder to comprehend.

On line 2 the statement `relate res to self across R4` could have been translated as *relate res to self*. But what does it mean that two objects are related? From an object-oriented view it means that they can access each other's public attributes and operations. The translation tries to capture this without going into details about the fundamentals of object-oriented design, substituting the reference `self` for the definite form of the class name of the referent, *the BookingProcess.*

The Action code finishes by sending a signal across the interface to the User component. Depending on if a room was found or not different signals are sent. Here the name of the interface, `HotelInterfaces` is substituted for the more informative `User` which is found by traversing the metamodel across the interface and its ports to the receiving component.

The signals exemplify a challenge for generating summarisations; should the parameters be translated using the parameter name, its defining expression or both? In the case of the `message` the expression is more descriptive than the name but for the `room:room_number` parameter both name and expression would be useful. The value of the `process_id` is less informative than the parameter name (`process_id` is included as a parameter to ensure that the right instance of

`BookingProcess` gets the reply from the User). To make an informed decision on the best phrasing in each case would require a semantic analysis of the values of the parameter expressions in comparison to the parameter names, something that is not supported by the transformation language.

4 DISCUSSION

4.1 Changing the Language

Different stakeholders have different needs in terms of the content of the summarisations, e.g. the developers want a quick introduction to the functionality of the system (Sridhara et al., 2010) while domain experts want to validate that certain requirements are met and maintained (Arlow et al., 1999). This can be accommodated by using different transformation rules for generating the grammars. One transformation can then generate a grammar that produces summarisations for the developers while another transformation is aimed towards the needs of the domain experts. The result is a shared abstract syntax that is realised by different concrete syntaxes to fit their respective needs using different functions from the RGL.

Some stakeholders might prefer another language than English. This can be facilitated by the multilingual aspect of the Grammatical Framework. In this approach the lexicon (or domain vocabulary) of the grammar is generated from the Action language. However, it is not obvious that the domain concepts share their names across languages. There are two ways to overcome this challenge; The naïve way is to ensure that the modelling elements use the terminology of the desired target language, by this approach the lexicon is automatically generated in the desired language. The other solution is to manually develop a lexicon per desired language, as explained in (Angelov and Ranta, 2009). Since the abstract functions defined by the RGL are language-independent, the same rules can be used for all desired languages. In this way the structure and content of the texts are preserved but with language-specific implementations of the sentences.

It is important to remember that any changes to the grammars are made through the transformation rules. As a consequence the transformation experts need to know the grammar that is used to model the texts well enough to implement the changes. It also means that neither the software modellers nor the customers need to know how the text is generated or how to formulate model transformations. When the transformations have been defined the translations are generated by a push on the button. The generation can then be repeated and reused for all models that conform to the same metamodel as the transformation rules (Atkinson and Kuhne, 2003; Mellor et al., 2004).

4.2 The Complexity of Model Transformation

The complexity of the model transformations does not lie in the complexity of the transformation rules but in the complexity of the modelling language they are applied to (Jézéquel et al., 2012).

On the target end of the transformation a knowledge of linguistics in general and the grammar API is needed to utilise the different categories and functions of the grammar in an efficient way. The alternative to grammars would be to generate text straight from the models with the tedious work of making sure that there is congruence between the verbs and the noun phrases as well as taking care of aspects like *a reservation* but *an interface*.

4.3 Text vs Models

Another benefit of natural language translations of textual software models embedded in graphical model elements is that they enable using any preferred text editor for searching after concepts and actions that should be in the text. Different modelling tools have their own support for searching with different interfaces, learning how to use them all is a tall request on stakeholders (Arlow et al., 1999).

5 CONCLUSIONS

The proposed way of translating Action code differs from previous work on code summarisation in that the platform-independent models already have filtered away the non-functional properties of the software, leaving the functional properties exposed. In comparison to previous research on generating natural language texts from software models this is the first attempt to generate software behaviour from platform-independent code.

The PIM can be reused to generate a number of different platform-specific models that include the usage of different APIs, programming languages, connections to operative systems and deployment on hardware. Since, the functionality of the system is captured in the PIM so the generated text gives a natural language summary of the system's behaviour disregarding how this behaviour is implemented. This means that the generated text can be used across platforms and updated by re-generation whenever the PIM is changed to reflect new requirements or bug-fixing. So, instead of having one framework for translating Java, another framework for translating C and a third for C++, a general framework for translating platform-independent code can be reused across platforms independently of how the system is realised.

6 FUTURE WORK

The mapping rules that define the transformation from PIM to PSM add the non-functional features that determine a certain combination of platform-specific details. Generated summarisations from the mappings could then describe the different profiles and properties of the system, such as safety and persistency.

The challenges in natural language generation from the combination of textual and graphical models is an interesting step to further explore. A case study is planned for including transformation rules that map the structure of the statemachines on to the generated translations. In this way the translations will give an overall structure of the software that follows the lifecycles of the system's classes and objects.

ACKNOWLEDGMENTS

This work was partially funded by the National Graduate School of Language Technology in Sweden and the Center of Language Technology in Gothenburg. The authors would like to thank Toni Siljamäki for sharing his insights in model transformations.

REFERENCES

Angelov, K. (2011). *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden.

Angelov, K. and Ranta, A. (2009). Implementing Controlled Languages in GF. In *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009*, volume 5972 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag.

Arlow, J., Emmerich, W., and Quinn, J. (1999). 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. Springer-Verlag.

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

Burke, D. A. and Johannisson, K. (2005). Translating Formal Software Specifications to Natural Language. In Blache, P., Stabler, E. P., Busquets, J., and Moot, R., editors, *LACL*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66. Springer Verlag.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2:137–167.

Clark, P., Murray, W. R., Harrison, P., and Thompson, J. A. (2009). Naturalness vs. Predictability: A Key Debate in Controlled Languages. In *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009*, volume 5972 of *Lecture Notes in Computer Science*, pages 65–81. Springer Verlag.

Forward, A. and Lethbridge, T. C. (2008). Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals. In *Proceedings of the 2008 international workshop on Models in Software Engineering*, MiSE '08, pages 27–32, New York, NY, USA. ACM.

Hähnle, R., Johannisson, K., and Ranta, A. (2002). An Authoring Tool for Informal and Formal Requirements Specifications. In Kutsche, R.-D. and Weber, H., editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248. Springer.

Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. (2010). On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In Antoniol, G., Pinzger, M., and Chikofsky, E. J., editors, *WCRE*, pages 35–44. IEEE Computer Society.

Heldal, R. and Johannisson, K. (2006). Customer Validation of Formal Contracts. In *OCL for (Meta-)Models in Multiple Application Domains*, pages 13–25, Genova, Italy.

Jézéquel, J.-M., Combemale, B., Derrien, S., Guy, C., and Rajopadhye, S. (2012). Bridging the Chasm Between MDE and the World of Compilation. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–17.

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

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

Larman, C. (2004). *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.

Ljunglöf, P. (2011). Editing Syntax Trees on the Surface. In *Nodalida'11: 18th Nordic Conference of Computational Linguistics*, volume 11, Riga, Latvia. NEALT Proceedings Series.

Mccarthy, J. (1962). Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, pages 21–28. North-Holland.

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

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

Mens, T. and Gorp, P. V. (2006). A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142.

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

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

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

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

Ranta, A. (2008). Grammars as software libraries. In Huet, G., Plotkin, G., Lévy, J.-J., and Bertot, Y., editors, *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press.

Ranta, A. (2009). The GF Grammar Resource Library. *Linguistic Issues in Language Technology*, 2(2).

Ranta, A. (2011). *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.

Rastkar, S., Murphy, G. C., and Bradley, A. W. J. (2011). Generating natural language summaries for crosscutting source code concerns. In *ICSM*, pages 103–112. IEEE.

Reiter, E. and Dale, R. (1997). Building applied natural language generation systems. *Nat. Lang. Eng.*, 3:57–87.

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

Spreeuwenberg, S., Van Grondelle, J., Heller, R., and Grijzen, G. (2010). Design of a cnl to involve domain experts in modelling. In Rosner, M. and Fuchs, N., editors, *CNL 2010 Second Workshop on Controlled Natural Languages*. CEUR Workshop Proceedings.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., and Vijay-Shanker, K. (2010). 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. ACM.

Sridhara, G., Pollock, L., and Vijay-Shanker, K. (2011). 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. ACM.

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

Stevens, P. (2007). A Landscape of Bidirectional Model Transformations. In Lämmel, R., Visser, J., and Saraiva, J., editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer.

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

Wyner, A., Angelov, K., Barzdins, G., Damljanovic, D., Davis, B., Fuchs, N., Hoefler, S., Jones, K., Kaljurand, K., Kuhn, T., Luts, M., Pool, J., Rosner, M., Schwitter, R., and Sowa, J. (2010). On controlled natural languages: Properties and prospects. In Fuchs, N. E., editor, *Proceedings of the Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *Lecture Notes in Computer Science*, pages 281–289, Berlin / Heidelberg, Germany. Springer Verlag.