Enabling Interface Validation through Text Generation

Håkan Burden, Rogardt Heldal and Peter Ljunglöf
Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
E-mail: burden@cse.gu.se, heldal@chalmers.se, peter.ljunglof@cse.gu.se

***Abstract*—To obtain the information encoded in software it is necessary to master both the implementation languages and the tools. This is not only a problem for managers and user groups who have a claim in the outcome but not the necessary training in software development to decode the implementation - our case study shows that it is also a problem for the software developers. In contrast, text can be understood by everyone. The generation of textual summaries from software artifacts enables a more accessible format for validating software. Based on findings from interviewing practitioners in industry, we have developed a prototype for generating natural language summaries from component interfaces for validation purposes.**

Keywords*—*Natural language processing; Reverse engineering; Software components;

I. Introduction

One way of handling complexity in large-scale software development is to decompose the system into autonomous subsystems that can be independently developed and maintained. In order to successfully integrate the implemented subsystems into a complete and well-functioning system it is necessary to define the connecting points, the interfaces, of the subsystems before or during the implementation [1], [2].

However, the validation of the interfaces is not trivial. For code-centric development it requires an understanding of the used programming languages. In a model-based approach to software development the problem is described by Arlow et. al. [3] as consisting of three main challenges; the necessity to understand the modeling tools used during development, the need to understand and interpret the models that describe the subsystems and their interfaces as well as the underlying paradigm of the models. So, independent of how the subsystems and their interfaces are implemented their validation can only be done by those who understand the implementation. This excludes many of those that have a claim in the delivered system, such as managers and user groups but it also affects many of the system developers. In contrast, textual summaries have the benefit that they can be consumed by all with a claim in the developed software [4].

In an on-going study of model-driven software development for embedded systems in large corporations we discovered that the software engineers had problems with accessing the information encoded in the models in general as well as verifying the correctness of the interfaces in particular. To investigate the effort needed to generate textual summaries from the interfaces we developed a prototype solution, reusing existing software models.

Contribution: First we describe the problem of validating interfaces according to practitioners in industry and why NLG is a possible solution. We then show that the generation of textual summarisations of interfaces can be done with a limited additional effort. The natural language generation technique can be used for other interface specifications that belong to the same modelling language.

Overview: The next section presents the theoretical context of our study, while the practical details are given in Section III. The results are found in Section IV and we finish off with a discussion and possibilities for future explorations in Section V.

II. Theoretical Context

We begin by explaining a few theoretical aspects concerning software models and the specific methodology that we used for generating textual summaries. Related contributions in the area of generating textual summaries from software conclude this section.

A. Model-Driven Engineering

One of the aims of using software models is to raise the level of abstraction in order to capture what is generic about a solution. Such a generic, or *platform-independent* [5], [6], solution can be reused for describing the same software independent of the platform i.e. the operating system, hardware and programming languages that are chosen to implement the system. The level of detail in the models then depend on if they are to be used as *sketches* giving the general ideas of the system, *blueprints* for manual adaptation into source code or if they are *code generators* and developed with a tool that supports the automatic *model translation* into source code [7]. A condition for the latter is that there is a *metamodel* [5], [8] that specifies the syntactical properties of the modelling language, just as textual programming languages like Java and C have a syntactical specification that can be encoded in BNF [9], [10].

B. Executable and Translatable UML

Executable and Translatable UML, xtUML; [11], [12], evolved from merging the Schlaer-Mellor methodology [13] with the Unified Modeling Language, UML. Three kinds of

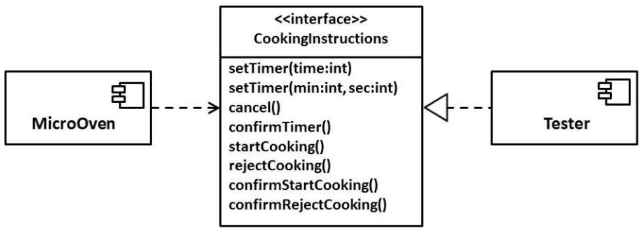


Fig. 1. The possible signals between the MicroOven and the Tester listed as an interface.

graphical diagrams are used together with a textual *Action language*. The diagrams are *component diagrams*, *class diagrams* and *state machines*. The diagrams are organised hierarchically so that state machines are found inside classes, classes inside components and components can be recursively nested inside other components. The Action language is then used within the diagrams to specify their behaviour and properties. When the models are complete with respect to behaviour and structure they can be automatically transformed into source code through *model translation*.

Components: A component interacts with other components across an interface. An interface declares a contract in form of a set of public features and obligations but not how these are to be implemented. The information and behaviour of the component is only accessible through the specified interface so that the component can be treated as a black box. An example of two components and their interfaces is shown in Figure 1. It consists of two components, MicroOven and Tester where MicroOven provides an interface which Tester depends on.

Class diagrams: For the case of this study it is sufficient to view an xtUML class diagram as equivalent to a UML class diagram.

State machines: In the context of xtUML, a state machine is used to model the lifecycle of a class or an object [13]. The transitions between the states can either be defined as internal events or the signals defined by the interface can be mapped onto the transitions so that the external calls change the internal state of the component.

Figure 2 shows the lifecycle of a test object residing inside the Tester component. From the state *Initial* it is possible to reach the *GenerateTimer* state by the internal trigger *next()*. When the external trigger *confirmTimer()* is called the test object is updated and the new state is *ValidateTimer*. Some of the states include pseudo-code to indicate the action to be taken when entering that state.

The semantics of xtUML state machines differ from that of finite-state automata in that the former can interact with their environment as by creating and deleting instances of classes, dispatching events in other state machines and trigger the sending of signals across interfaces etc. Also, if a trigger event does not enable a transition it is not necessarily an error since transition triggers can be ignored if so desired.

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

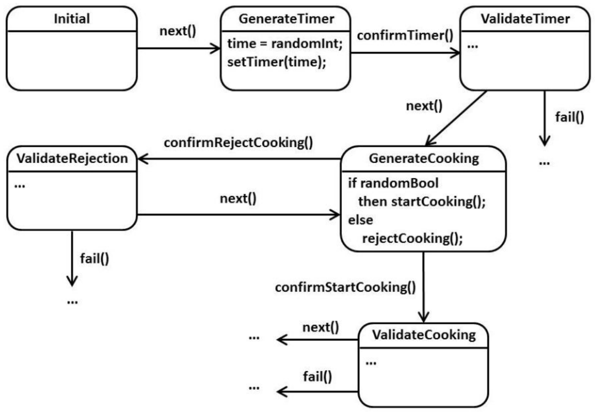


Fig. 2. The state machine in Tester describing the validation process.

metamodel [13]. Since the Action language shares the same metamodel as the graphical models it can be used to define how values and class instances are manipulated as well as how the classes change their state. Thus we can find Action language within the operations of the classes but it is also used to define the behaviour and the flow of calls through the interface between the components.

Model translation: Code generation is a specific case of model translation with the aim of translating the model into code. However, model translations can just as well be used for reverse engineering the model into more abstract representations [14], [15]. Model translations are defined according to the metamodel, enabling the same transformations to be reused across domains [5], [16], just like a C compiler is defined on the BNF grammar, not on a specific C program [9]. The models become the code. An example of what the xtUML transformation rules look like and how they can be used is found in Figure 5, which will be further explained in section III-D.

C. Related Work

Previous research has reported on both formal and informal ways of validating the behaviour and structure of software. Examples of formal methods for validating the interfaces are presented by Hatcliff et. al. [17], Mencl [18] as well as Uchitel and Kramer [19] among many. However, they all have similar problems as those mentioned previously by Arlow et. al. [3]; formal methods require knowledge of the tools, knowledge of the used models and their paradigm as well as knowledge of the formal methods.

Lately there has been an increase in the attention towards more informal possibilities for validating software. Spreeuwenberg et. al. [20] argue that if you want to include all stakeholders in the development process you need to have a textual representation of the software models that has the right level of abstraction. In their case they generate a controlled natural language [21] to validate candidate policy decisions for the Dutch Immigration Office.

Another approach towards text generation from platform-independent representations is the translation between the Object Control Language, OCL [22], and English [23], [24].

This work was followed up by a study on natural language generation of platform-independent contracts on system operations [25], where the contracts were defined as OCL constraints that specified the pre- and post-conditions of system operations, i.e. what should be true before and after the operation was executed.

A crosscutting concern is a piece of functionality, such as an algorithm, that is implemented in one or more components. As a result of being scattered across the implementation they are difficult to analyse and when changed or updated it is difficult to estimate how the changes are going to affect the rest of the implementation. Rastkar et. al. [26] argue that having a natural language summary of each concern enables a more systematic approach towards handling the changes. They have therefor implemented a system for generating summaries in English from Java implementations.

Sridhara et. al. [27], [28] have also generated natural language from software implementations, in their case from Java code. The motivation is that understanding code is both a time consuming activity and that accurate descriptions can summarise the algorithmic behaviour of the code and as well as 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 software synchronized. Another approach to textual summarisations of Java code is given by Haiduc et. al. [29]. 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 two previous publications on generating textual descriptions from xtUML. The first describes how natural language specifications can be generated from class diagrams [30] while the second reports on the translation from Action language to English [31] e.g. these publications concentrate on generating textual summaries that describe the internal properties of the components instead of the interaction among components.

III. Case Study

In our collaboration on model-driven engineering with Ericsson AB, Volvo Group Trucks Technology and Volvo Cars Corporation we encountered the problem of validating component interfaces. During our interviews the engineers reported that it was sometimes challenging to validate that the interface was correctly implemented and that the information needed for the validation could be difficult to obtain. As a response we developed a prototype to explore the possibilities to generate natural language summaries for validating component interfaces while keeping the added effort to a minimum.

A. Motivation

The interviews were conducted in January 2013, stretching into April 2013. The following interview extract illustrates the problem of understanding the implementation by reading its textual specification.

But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

Another issue is that sometimes the engineers are asked to specify the interface before they fully understand the internal behaviour of the component being developed. This means that defining the interface becomes guess work and subsequently there are signals that will never be used but still be given their share of the limited processing capacity.

Q: So do you overload the interface? Throw in a signal just in case?

A: Yes, that is what we do. At least I do it [. . .] and then you end up with the problem knowing which signal it is you should actually use.

One of the other interviewees had developed a work-around for handling that the interface specification was constantly outdated. The solution is to sieve through a second document after the information that concerns the interface being developed and translate that information into a new, temporary, specification.

We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked. What do I need? I need this and this. So, yeah, so I do it manually, I guess.

As a final example of the problems concerning the validation of the component interfaces, a software architect stated that the development tools were difficult to learn and that the development process would be much smoother if there was an accurate textual description of the implementation.

The tools are too unintuitive [. . .] the threshold for learning how to use them is high [. . .] but everybody knows how to consume text.

B. Aim

Generating text from the implementation should be a suitable solution since it allows the textual description to always be consistent with the implementation as well as understandable by all those with a claim in the project.

The aim of the text generation is a textual description of the intended usage of the interface with as little added effort as possible. For the generation to be feasible in an industrial setting it is beneficial if the generation rules can be maintained and updated without requiring new skills of the engineers. At the same time, the reuse of existing artifacts for generating the summaries will decrease their cost.

Two paragraphs are included in the generated text, one for the intended usage of the interface and one for the unused signals of the interface.

C. Setup

The generation is possible due to the reuse of an existing test model, that was adapted from Heldal et. al. [32]. They developed an executable test model for a microwave oven, as illustrated in Figure 1. The test model is designed to capture the intended dialogue between the MicroOven and the user, here

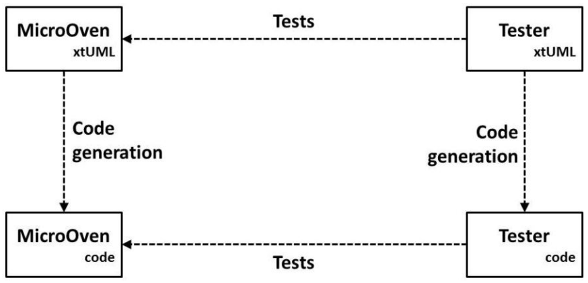


Fig. 3. The Tester-model can be reused at code level through code generation.

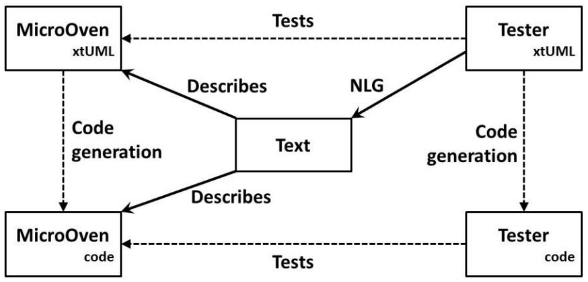


Fig. 4. The generated text describes both the xtUML model and the generated code.

represented by the Tester component, as well as its possible error states and constraints. The sequence of the states and transitions therefor follows the process of the MicroOven, with additions for handling erroneous interactions. After the test case is initialised the test-pattern is to generate a signal to the MicroOven across the interface with random values for each parameter. The MicroOven's response is then validated before the Tester transitions into the *next* state in order to generate a new signal with random values. The test case needs to be able to store the results of prior interactions in order to compute the expected value in the validation states and compare that to the given response. If there is a mis-match the test case generates an internal *fail()* event and stores the resulting state so it can be diagnosticised.

After the MicroOven-model has been validated using the Tester-model they are both translated into platform-dependent source code. The MicroOven is then tested again, now as code by the Tester-code, to see that the intended behaviour of the oven remains the same after deployment. The relationship between the different representations of MicroOven and Tester are depicted in Figure 3.

D. Text Generation

Figure 5 shows a fragment of the generation rules. The rules are defined using the Rule Specification Language which are integrated with the xtUML tools [11]. On the first row the signals defined by the `interface` are selected by traversing the concepts of the metamodel according to their relationships. The concept `C_EP` refers to the executable properties of the interface and `C_AS` refers to those executable properties that are signals. The relationships between the concepts are referred to by the unique names `R4003` and `R4004`. Rows 3 and 4 show how the generated text is going to be physically represented [33] as html-pages, using a table since it enables

```
01 .select many definedSignals related by interface -> C_EP[R4003] -> C_AS[R4004]
02 [...]
03 <table border="0">
04 <hr>Unused signals in MicroOven:</hr>
05 .assign unusedSignals = definedSignals - usedSignals
06 .if (not_empty unusedSignals)
07   .for each signal in unusedSignals
08     .invoke paramText = GetParamData
09     <tr><td><i>${signal.name}(paramText)</i></td></tr>
10   .end for
11 .else
12 <b>All defined signals are used.</b>
13 .end if
14 </table>
```

Fig. 5. An example of a model-to-text transformation using xtUML.

the representation of parallel success paths. All rows that start with a punctuation mark are statements defined by the transformation language while those rows that do not start with punctuation mark define the generated text. The string value of a variable v is obtained by getting its literal text value, `$`$\{v\}$, as in row nine where the signal's name is inserted into the table. Even if the success story only includes those signals that are implemented in the intended usage of the MicroOven the variable `usedSignals` on row five is defined by traversing the entire state machine in order to collect all signals that are used to implement the test case. On row eight the parameters are converted into a textual representation, `paramText`, by calling the function `GetParamData`, which is defined by the translation engineer.

In the context of our study it is not relevant to mention in what class the state machine resides that handles the interaction across the interface. That information is excluded in the content selection phase [34] since it is the possible interaction across the interface, as modelled by the state machine, that is interesting, not the internal structure of the components.

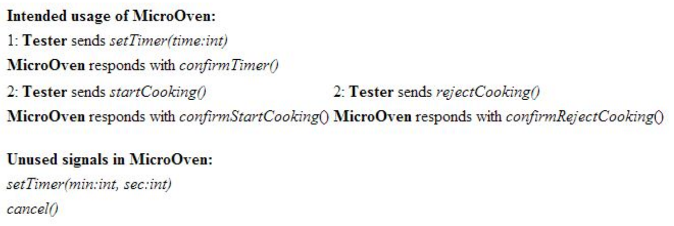
For the success path the structure of the text follows the order imposed by the transitions of the state machine, only considering the names of the transitions that constitute the intended usage.

IV. Results

The algorithm for navigating through the metamodel to generate the textual summaries is on the size of 100 statements. In comparison, a model compiler for generating Java programs consists of 500.000 statements but that covers the entire xtUML definition. Since the state machines and the Action language are so intertwined with the interfaces it is not possible to get a number for the statements needed for translating the interfaces as such into Java. The number of statements for the textual generation is dependent on the present content selection and will increase if more model concepts are to be present in the generation.

In Figure 6, an example of a generated text is shown. It depicts the summarisation of the interface in Figure 1 as implemented by the state machine in Figure 2. The top-half of the web page shows the intended usage of the oven and the bottom-half details which signals in the interface that are unused in the implementation. The intended usage is given in a table format where alternative usages are given on the same row, side-by-side.

The name of the interacting component is Tester, which is carried on to the generated text. This emphasizes that naming



Intended usage of MicroOven:
1: **Tester** sends *setTimer(time:int)*
MicroOven responds with *confirmTimer()*
2: **Tester** sends *startCooking()* 2: **Tester** sends *rejectCooking()*
MicroOven responds with *confirmStartCooking()* **MicroOven** responds with *confirmRejectCooking()*

Unused signals in MicroOven:
setTimer(min:int, sec:int)
cancel()

Fig. 6. An example of a textual summary.

conventions will affect the readability and understanding of generated texts when they are derived from software implementations. In this case the reading of the generated text would have benefited of naming the testing component to User, who it is meant to represent.

The paragraph for unused signals include *setTimer(min:int, sec:int)*, which represents how the interface was overloaded at the point of specification due to the fact that it was unclear how the MicroOven would be used. Since then the decision was taken to specify times in seconds only but the interface was not changed to reflect this decision. The generated text clearly identifies that there is a mismatch between the specification of the interface and its implementation.

Since the text is automatically generated from the source model it is possible to have a text generated that is consistent with the implementation whenever it is needed; e.g. when considering the implications of adding new functionality, to validate that new functionality conforms to the requirements during implementation or after implementation to understand how the software is intended to be used. This is shown in Figure 4 where the generated text can be used both to describe the original model or the implementation at code-level.

When the model is translated into code the information enclosed in the model is extended with details specific to operative system, chosen programming languages etc. in order to enable the deployment of the generated source code on a specific platform. This added information is then automatically excluded in the generated text since it is not present in the model. The benefit is that the generated text automatically becomes a summary of the interface that focuses on its intended usage while it abstracts away from how the behaviour is obtained. The text can then be reused independent on how the interface is realised on different platforms. As an example, the position of the signal *setTimer(time:int)* in the sequence of intended interactions between the oven and the user does not depend on if C or Java is used to realise the interface.

The algorithms for generating the success stories and to document unused signals are defined upon the xtUML metamodel. This means that they are reusable across different models that adhere to the metamodel, just as a compiler is defined upon the BNF grammar of programming language and therefor reusable across programs [9].

The structure and naming of concepts and relationships in the metamodel is the main source of complexity in this approach to NLG. Knowing what the concepts and relationships refer to is more challenging than how to map them into a textual representation.

V. Discussion and Future Work

The Object Management Group are the owners of the UML specification and the architects behind the MDA [5], [6] approach to using UML for software development. Their approach for defining the sequencing of the interface signals is to develop a new model, a protocol state machine[35]. Their solution results not only in an additional effort of developing a new model for explaining an old one, but also relies on the same techniques that made it difficult to verify the old model in the first place. As an additional contribution we show how an existing test model can be used for the same purpose as a protocol state machine as well as the source for an NL summary explaining the protocol of the interface.

In relation to previous work on text generation from xtUML our approach does not rely on the understanding of complex linguistic tools. The benefit of only using the same techniques for NLG as for code generation is that there is no additional training cost for companies. This makes it easier to adopt NLG in an industrial context since the number of software engineers with an understanding of both metamodelling and language technology are few. The mapping of metamodel concepts and relationships into linguistic properties will increase the complexity of macro- and microplanning. The drawback of our approach is the limited expressiveness of the transformation rules. For a more varied text structure, less repetitive sentences or for languages with a richer morphology it would be necessary to apply an NLG approach that incorporates linguistic competence. However, striking the balance between richer NLG and what companies are prepared to invest in hiring new competence is yet to be investigated. Our hope is that we will be able to start a new collaboration to enable practitioners in industry to evaluate both the generated texts and the generation procedure. Both lines of query would help to better understand where the balance between cost and readability lies.

Due to the time constraints of the MDE study there was not enough time to gain access to the original models to generate documentation within the industrial context where the issues were found. Instead, a prototype was implemented to show how the documentation could be generated from software models with a minimal effort. This opens for another possible route for the future, to further explore the possibilities of NLG for validation purposes in an industrial context. As seen in the related literature there is little involvement from industry to actually use NLG for validation purposes. We believe that this contribution can be a first step to address the problems that engineers actually are facing and as such also open for new ways of adapting NLG and summarisation techniques to the engineer's needs and context.

Acknowledgment

The authors would like to thank the Graduate School in Language Technology, the Center of Language Technology and the Software Center for funding and support as well as the contact persons at respective company for facilitating the study.

References

[1] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[2] G. T. Heineman and W. T. Councill, Eds., *Component-based software engineering: putting the pieces together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

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

[4] D. Firesmith, "Modern Requirements Specification," *Journal of Object Technology*, vol. 2, no. 2, pp. 53–64, 2003.

[5] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," Object Management Group, Tech. Rep., 2003.

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

[7] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[8] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36 – 41, September 2003.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston: Pearson Education, Inc., 2007.

[10] M. Wimmer and G. Kramler, "Bridging Grammarware and Modelware," in *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed. Springer Berlin / Heidelberg, 2006, vol. 3844, pp. 159–168.

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

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

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

[14] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, March 2006.

[15] P. Stevens, "A Landscape of Bidirectional Model Transformations," in *Generative and Transformational Techniques in Software Engineering II, International Summer School*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235. Braga, Portugal: Springer, July 2007, pp. 408–424.

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

[17] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proceedings of the 25th International Conference on Software Engineering*, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. Portland, Oregon, USA: IEEE Computer Society, May 2003, pp. 160–173.

[18] V. Mencl, "Specifying Component Behavior with Port State Machines," *Electronic Notes in Theoretical Computer Science*, vol. 101, pp. 129–153, 2004.

[19] S. Uchitel and J. Kramer, "A workbench for synthesising behaviour models from scenarios," in *Proceedings of the 23rd International Conference on Software Engineering*, H. A. Müller, M. J. Harrold, and W. Schäfer, Eds. Toronto, Ontario, Canada: IEEE Computer Society, May 2001, pp. 188–197.

[20] S. Spreeuwenberg, J. Van Grondelle, R. Heller, and G. Grijzen, "Design of a CNL to Involve Domain Experts in Modelling," in *CNL 2010 Second Workshop on Controlled Natural Languages*, M. Rosner and N. Fuchs, Eds. Springer, 2010, pp. 175–193.

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

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

[23] R. Hähnle, K. Johannisson, and A. Ranta, "An Authoring Tool for Informal and Formal Requirements Specifications," in *FASE 2002, Fundamental Approaches to Software Engineering, 5th International Conference*, ser. Lecture Notes in Computer Science, R.-D. Kutsche and H. Weber, Eds., vol. 2306. Grenoble, France: Springer, April 2002, pp. 233–248.

[24] D. A. Burke and K. Johannisson, "Translating Formal Software Specifications to Natural Language," in *5th International Conference on Logical Aspects of Computational Linguistics*, ser. Lecture Notes in Computer Science, P. Blache, E. P. Stabler, J. Busquets, and R. Moot, Eds., vol. 3492. Bordeaux, France: Springer Verlag, April 2005, pp. 51–66.

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

[26] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *27th International Conference on Software Maintenance*. Williamsburg, VA, USA: IEEE, September 2011, pp. 103–112.

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

[28] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 101–110.

[29] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *WCRE*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 35–44.

[30] H. Burden and R. Heldal, "Natural Language Generation from Class Diagrams," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa 2011. Wellington, New Zealand: ACM, October 2011.

[31] ——, "Translating Platform-Independent Code into Natural Language Texts," in *MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

[32] R. Heldal, D. Arvidsson, and F. Persson, "Modeling Executable Test Actors: Exploratory Study Done in Executable and Translatable UML," in *19th Asia-Pacific Software Engineering Conference*, K. R. P. H. Leung and P. Muenchaisri, Eds. Hong Kong, China: IEEE, December 2012, pp. 784–789.

[33] J. Bateman and M. Zock, "Natural Language Generation," in *The Oxford Handbook of Computational Linguistics*, ser. Oxford Handbooks in Linguistics, R. Mitkov, Ed. Oxford University Press, 2003, ch. 15.

[34] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[35] "OMG Unified Modeling Language^{TM}(OMG UML), Superstructure," http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/, version 2.4.1. Accessed August 2013.