Thesis for the Degree of Doctor of Philosophy

A Scholarship Approach to Model-Driven Engineering

Håkan Burden

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2014

A Scholarship Approach to Model-Driven Engineering
ISBN 78-91-628-9097-1
Copyright © Håkan Burden, 2014

Technical Report no. 114D
Department of Computer Science and Engineering
Research groups: Language Technology and Model-Driven Engineering

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

Typeset with LaTeX using GNU Emacs
Printed at Chalmers University of Technology, Gothenburg, 2014

"It's the side effects that save us"

The National

Abstract

Model-Driven Engineering is a paradigm for software engineering where software models are the primary artefacts throughout the software life-cycle. The aim is to define suitable representations and processes that enable precise and efficient specification, development and analysis of software.

Our contributions to Model-Driven Engineering are structured according to Boyer's four functions of academic activity – the scholarships of teaching, discovery, application and integration. The scholarships share a systematic approach towards seeking new insights and promoting progressive change. Even if the scholarships have their differences they are compatible so that theory, practice and teaching can strengthen each other.

Scholarship of Teaching While teaching Model-Driven Engineering to undergraduate students we introduced two changes to our course. The first change was to introduce a new modelling tool that enabled the execution of software models while the second change was to adapt pair lecturing to encourage the students to actively participate in developing models during lectures.

Scholarship of Discovery By using an existing technology for transforming models into source code we translated class diagrams and high-level action languages into natural language texts. The benefit of our approach is that the translations are applicable to a family of models while the texts are reusable across different low-level representations of the same model.

Scholarship of Application Raising the level of abstraction through models might seem a technical issue but our collaboration with industry details how the success of adopting Model-Driven Engineering depends on organisational and social factors as well as technical.

Scholarship of Integration Building on our insights from the scholarships above and a study at three large companies we show how Model-Driven Engineering empowers new user groups to become software developers but also how engineers can feel isolated due to poor tool support. Our contributions also detail how modelling enables a more agile development process as well as how the validation of models can be facilitated through text generation.

The four scholarships allow for different possibilities for insights and explore Model-Driven Engineering from diverse perspectives. As a consequence, we investigate the social, organisational and technological factors of Model-Driven Engineering but also examine the possibilities and challenges of Model-Driven Engineering across disciplines and scholarships.

Acknowledgments

I want to start by thanking my supervisors – Rogardt Heldal, Peter Ljunglöf and Tom Adawi. I am indebted to their inspiration and patience. I likewise want to acknowledge the various members of my PhD committee at Computer Science and Engineering – Aarne Ranta, Bengt Nordström, Robin Cooper, David Sands, Jan Jonsson, Koen Claessen and Gerardo Schneider. Thanks!

There are three research environments that I particularly want to mention. The first is the Swedish National Graduate School of Language Technology, GSLT, for funding my graduate studies. The second research environment is the Center for Language Technology at the University of Gothenburg, CLT, which has funded some of the traveling involved in presenting the publications included in the thesis. I was also very fortunate to receive a grant from the Ericsson Research Foundation that enabled me to present two of the publications.

Over the years I have met far too many people at conferences and workshops, in classrooms and landscapes to mention you all – I appreciate the talks we had in corridors and by the coffee machine. I have also had some outstanding room mates over the years. It's been a pleasure sharing office space with you. A special thank you to the technical and administrative staff who have made my academic strife so much easier.

There are some researchers and professionals in the outside world that deserve to be mentioned; Joakim Nivre, Leon Moonen, Toni Siljamäki, Martin Lundquist, Leon Starr, Stephen Mellor, Staffan Kjellberg, Jonn Lantz, Dag Sjøberg, Jon Whittle, Mark Rouncefield, John Hutchinson – as well as all my past and present students. Cheers to the engineers at Ericsson, Volvo Cars and Volvo Trucks that had so many insights to share and patience with my constant probing. And to (nearly) all anonymous reviewers – Thanks!

On the private side I want to thank Ellen, Malva, Vega, Tora and Björn for providing an alternative reality. And a big thanks to my numerous friends and relatives who keep asking me what I do for a living. To all my neighbors at Pennygången. Your encouragement and support has meant a lot.

Associates, Friends, Family, Relatives and Neighbours[1] – You've all helped me to become the scholar I want to be. Thank You!

Håkan Burden
Gothenburg, 2014

[1]Feel left out? I do hope I can compensate you with a free copy of my thesis!

Included Publications

Scholarship of Teaching

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

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Model Modelling and Encourage Active Learning. In *Proceedings of ALE 2012, 11th Active Learning in Engineering Workshop*, Copenhagen, Denmark, June 2012.

Scholarship of Discovery

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.

Håkan Burden and Rogardt Heldal. Translating Platform-Independent Code into Natural Language Texts. In *Proceedings of MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

Scholarship of Application

Håkan Burden, Rogardt Heldal, and Martin Lundqvist. Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing. In *Proceedings of 6th International Workshop on Multi-Paradigm Modeling, MPM'12*, Innsbruck, Austria, October 2012. ACM.

Rogardt Heldal, Håkan Burden, and Martin Lundqvist. Limits of Model Transformations for Embedded Software. In *Proceedings of 35th Annual IEEE Software Engineering Workshop*, Heraklion, Greece, October 2012. IEEE.

Scholarship of Integration

Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering - Are the Tools Really the Problem? In *Proceedings of MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

Ulf Eliasson and Håkan Burden. Extending Agile Practices in Automotive MDE. In *Proceedings of XM 2013, Extreme Modeling Workshop*, pages 11-19, Miami, USA, October 2013.

Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Enabling Interface Validation through Text Generation. In *Proceedings of VALID 2013, 5th International Conference on Advances in System Testing and Validation Lifecycle*, Venice, Italy, November 2013.

Håkan Burden, Rogardt Heldal, and Jon Whittle. Comparing and Contrasting Model-Driven Engineering at Three Large Companies. In *Proceedings of ESEM 2014, 8th International Symposium on Empirical Software Engineering and Measurement*, Torino, Italy, September 2014.

Additional Publications

The following publications are also the result of my time as a PhD student. The majority are peer-reviewed short papers or extended abstracts and therefor not included in the thesis. The exceptions are a technical report and a popular science contribution (as indicated when appropriate).

Håkan Burden, Rogardt Heldal, and Tom Adawi. Assessing individuals in team projects: A case study from computer science. Presented at *Conference on Teaching and Learning - KUL*, Gothenburg, Sweden, January 2011.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Students' and teachers' views on fair grades - is it possible to reach a shared understanding? In proceedings of *3:e Utvecklingskonferensen för Sveriges ingenjörsutbildningar*, Norrköping, Sweden, 2011.

Håkan Burden. *Three Studies on Model Transformations - Parsing, Generation and Ease of Use.* Licentiate thesis, University of Gothenburg, Gothenburg, Sweden, June 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Enhance Reflective Practice and Teacher Development. In *Proceedings of ISL2012 Improving Student Learning Symposium*, Lund, Sweden, August 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing - Catching that teachable moment. Published in *Lärande i LTH* (A Swedish popular science magazine), October 2012.

Håkan Burden, Jones Belhaj, Magnus Bergqvist, Joakim Gross, Kristofer Hansson Aspman, Ali Issa, Kristoffer Morsing, and Quishi Wang. An Evaluation of Post-processing Google Translations with Microsoft® Word. In *Proceedings of The Fourth Swedish Language Technology Conference*, Lund, Sweden, October 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing Model-Driven Software Development. Presented at *Conference on Teaching and Learning - KUL*, Gothenburg, Sweden, January 2013.

Giuseppe Scanniello, Miroslaw Staron, Håkan Burden, and Rogardt Heldal. *Results from Two Controlled Experiments on the Effect of Using Requirement Diagrams on the Requirements Comprehension.* Technical report, Research Reports in Software Engineering and Management, Chalmers University of Technology and University of Gothenburg, March 2013.

Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Opportunities for Agile Documentation Using Natural Language Generation. In *Proceedings of ICSEA 2013, 8th International Conference on Software Engineering Advances*, Venice, Italy, November 2013.

Giuseppe Scanniello, Miroslaw Staron, Håkan Burden and Rogardt Heldal. On the Effect of Using SysML Requirement Diagrams to Comprehend Requirements: Results from Two Experiments. In *Proceedings of EASE 2014, 18th International Conference on Evaluation and Assessment in Software Engineering*, London, UK, May 2014.

Håkan Burden and Tom Adawi. Mastering Model-Driven Engineering. In *Proceedings of ITiICSE 2014, 19th Annual Conference on Innovation and Technology in Computer Science Education*, Uppsala, Sweden, June 2014.

Håkan Burden. Putting the Pieces Together - Technical, Organisational and Social Aspects of Language Integration for Complex Systems. In *Proceedings of GEMOC 2014, 2nd International Workshop on The Globalization of Modeling Languages*, Valencia, Spain, September 2014.

Håkan Burden, Sebastian Hansson and Yu Zhao. How MAD are we? Empirical Evidence for Model-driven Agile Development. In *Proceedings of XM 2014, 3rd Extreme Modeling Workshop*, Valencia, Spain, September 2014.

Imed Hammouda, Håkan Burden, Rogardt Heldal and Michel R.V. Chaudron. CASE Tools versus Pencil and Paper – A student's perspective on modeling software design. In *Proceedings of EduSymp 2014, 10th Educators' Symposium colocated with MODELS 2014*, Valencia, Spain, September 2014.

Personal Contribution

In the case of *Natural Language Generation from Class Diagrams*, *Translating Platform-Independent Code into Natural Language Texts* and *Enabling Interface Validation through Text Generation* the idea of natural language generation from software models was conceived by Peter Ljunglöf while I was the main contributor in planning and conducting the research. I was also the main author of the publications while Rogardt Heldal and Peter Ljunglöf helped in giving the contributions their context.

For *Pair Lecturing to Model Modelling and Encourage Active Learning* I developed the idea of pair lecturing together with Rogardt Heldal while Tom Adawi played a crucial role in helping us evaluate and communicate the results. I was the main author.

Rogardt Heldal initiated the introduction of Executable and Translatable UML into our course. I was a main contributor in terms of planning, conducting, evaluating and writing *Executable and Translatable UML - How Difficult Can it Be?*.

The two papers *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* and *Limits of Model Transformations for Embedded Software* were initiated and conducted without my contribution. Here my contribution was in evaluating and presenting the outcome together with Rogardt Heldal and Martin Lundqvist.

In the writing of *Industrial Adoption of Model-Driven Engineering - Are the Tools Really the Problem?* my contribution was in validating the taxonomy based on the interviews I had conducted as well as contributing to the writing of the publication in general and being the main author of section 5.

Rogardt Heldal and I came up with the idea for *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* while Jon Whittle was instrumental in how to analyse and communicate the result. Jon Whittle and I wrote most of the text with Rogardt Heldal contributing to section 4. I am responsible for all data collection.

Ulf Eliasson was the main author of *Extending Agile Practices in Automotive MDE* except for section three to which we contributed equally. Since the publication is the synthesis of two independent studies both authors have a shared responsibility for data collection and evaluation.

Contents

Summary and Synthesis

Håkan Burden[1]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
2014

1 Introduction

1.1 Context – Raising the Level of Abstraction

In the beginning of software development the programs were manually specified by low-level instructions detailing how data at a specific memory location should be moved to a new location or how the data at one explicit location was the sum of the data at two other locations [1]. While the low-level instructions gave the programmer detailed expressions and full control it also meant that the programming was error prone, slow and updating the programs with new instructions was difficult. By automating some of the tasks – such as memory allocation – and combining repetitive sequences of low-level instructions to one high-level short-hand notation programming became more oriented towards the needs of the human users instead of the constraints of the machines.

In the early 1970's the trend towards more abstract representations had resulted in languages with so different ways of expressing the computations that they represented different paradigms. Examples of paradigms that stem from this era are imperative languages where C is still a major player, object-oriented languages such as Smalltalk and Java as well as functional languages like ML and Haskell. The evolution has continued and during the 1990's scripting languages like PHP and Perl saw the light.

1.2 Thesis Subject – Model-Driven Engineering

In continuation of the trend of raising the level of abstraction Model-Driven Engineering, MDE, emerged just after the turn of the century [65]. Models have been used for a long time in engineering disciplines, both to describe the present and to predict the future [100]. What distinguishes a model in the context of software development is that models are not just an aid to describe and predict – models can play a key role in implementation when used as programming languages [44]. By automatically transforming the abstract model into more concrete code MDE promises a means for handling the growing complexity of software as well as increasing the development speed and the quality of the software by automating tedious and error-prone tasks [80].

1.3 Thesis Contributions

This thesis reports on the possibilities and shortcomings of Model-Driven Engineering from three different settings, each with their own perspectives:

Education In order to improve our students' ability to develop models we introduced two changes to our course. The first change was to introduce a tool that enabled the models to be used as a programming language [27] and the second change was to introduce pair lecturing to encourage the students to actively participate in developing models during lectures [23].

Basic Research While models can be more abstract than the generated code it does not mean that they are necessarily easier to understand. We used existing technologies for transforming models into code to instead generate natural language representations from class diagrams [21], high-level behavioural descriptions of the models [22] and component interfaces [24]. The benefit of this approach in comparison to generating from code is that the texts are reusable across different implementations of the same model.

Industrial Practice Raising the level of abstractions might seem to be a technical issue but our results enforce how the success of adopting Model-Driven Engineering depends on both organisational and social factors as well as technical [26, 54, 119]. In the long run Model-Driven Engineering empowers new user groups to become software developers as well as speeding up the development process [28, 39].

1.4 Conceptual Framework – Scholarship

From the onset the research on MDE was carried out independently within each perspective – improving how to teach MDE to students, exploring the possibilities of model transformations for natural language generation and the adaption of MDE in an industrial setting. During the latter half of our PhD studies we recognised that these three perspectives had common features and in 2013 we identified how they coincided with Boyer's notion of scholarship [13, 14].

Boyer refers to basic research as the scholarship of *discovery*, education maps to the scholarship of *teaching* while the scholarship of *application* covers all interaction with non-academic organisations including industry. Boyer also introduces a fourth scholarship named *integration* which aims to seek synergies among disciplines and scholarships as well as new interpretations of scientific contributions or open research questions.

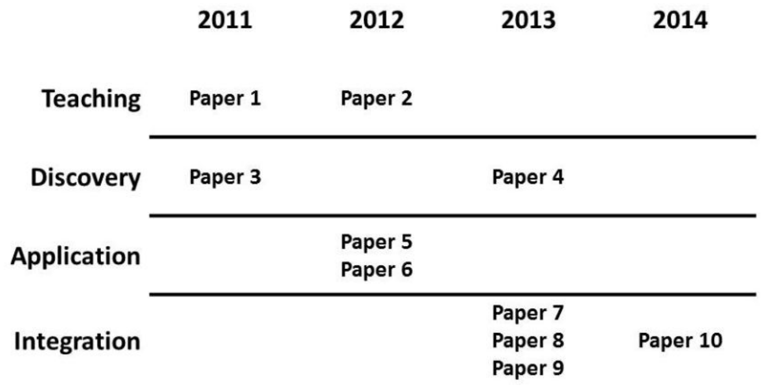
While it was originally unintentional, our way of approaching MDE corresponds to Boyer's definition of Scholarship. We will therefore use the four scholarships of Discovery, Teaching, Application and Integration to structure the contributions of the included publications but also to show how the originally independent research tracks have merged and influenced each other over time.

1.5 Thesis Overview

The remaining sections of this chapter are organised as follows :

Summary and Synthesis The next section will give more detail to the concepts of Model-Driven Engineering and scholarship. The used research methodologies are then described in section 3. The contributions of the included publications are given per scholarship in section 4. Section 5 discusses insights from combining a scholarship approach and graduate studies in terms of opportunities for reflection. The conclusion is found in section 6 while future research directions are presented in section 7.

The publications supporting the contributions are then included as chapters for each scholarship:



	2011	2012	2013	2014
Teaching	Paper 1	Paper 2		
Discovery	Paper 3		Paper 4	
Application		Paper 5 Paper 6		
Integration			Paper 7 Paper 8 Paper 9	Paper 10

Figure 1: The included publications – referenced by paper – organised according to scholarship and publication year.

Appendix A: Scholarship of Teaching The relevant publications are included as paper 1 *Executable and Translatable UML – How Difficult Can it Be?* [27] and paper 2 *Pair Lecturing to Model Modelling and Encourage Active Learning* [23].

Appendix B: Scholarship of Discovery Included publications are paper 3 *Natural Language Generation from Class Diagrams* [21] and paper 4 *Translating Platform-Independent Code into Natural Language Texts* [22].

Appendix C: Scholarship of Application Paper 5 was published as *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* [26] and paper 6 as *Limits of Model Transformations for Embedded Software* [54].

Appendix D: Scholarship of Integration The four included papers are Paper 7 *Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem?* [119], paper 8 *Extending Agile Practices in Automotive MDE* [39], paper 9 *Enabling Interface Validation through Text Generation* [24] and paper 10 *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* [28].

An overview of the included papers, sorted under scholarship together with their respective publication year is found in Figure 1.

2 Background

Before we go further into the contributions of the included publications, the paradigm of Model-Driven Engineering and the concept of Scholarship are described in more detail.

2.1 Model-Driven Engineering

Model-driven engineering is a paradigm for developing software relying on models – or abstractions – of more concrete software representations to not only guide the development but constitute the single source of implementation.

2.1.1 Software Models

There are a number of definitions regarding the qualities and usage of models for scientific purposes in general [29, 35, 107] and also more specifically for software engineering [33, 72, 74, 94, 99, 100]. In the context of this thesis we will emphasise two notions of software models; first the relationship between model and code, second how models can be used as the actual implementation language(s).

Figure 2 gives the spectrum of model and code as defined by Brown [17]. Brown defines the code as a running application for a specific runtime platform (acknowledging that the code itself is a model of bit manipulations). In relation to Brown's definition of code, a model is defined as an artifact that assists in creating the code by transformations, enables prediction of software qualities and/or communicates key concepts of the code to various stakeholders.

Code only To the farthest left of Figure 2 is the code only approach to software development, using languages such as C or Java but without any separately defined models. While this approach often relies on abstractions such as modularization and APIs there is no notion of modelling besides partial analysis on whiteboards, paper and slide presentations. This scenario is referred to by Fowler as using models for sketching [44].

Code visualization A step to the right, and towards more modelling, is code visualization. Code is still the only implementation level but different kinds of abstractions are used for documentation and analysis. While some abstractions are developed manually some are automatically generated from the code – such as class diagrams or dependencies between components and modules.

Roundtrip engineering The third possibility according to Brown is that code and model together form the implementation. One example would be a system specification at model level that is then used to automatically generate a code skeleton fulfilling the specification. The skeleton is then manually elaborated to add more functionality and details. As the code evolves the model is updated with the new information to ensure consistency between specification and implementation. The term round-trip engineering comes from the fact that models and code are developed in parallel and require a roundtrip from model to model via code (or vice versa).

Model-centric In model-centric development the model specifies the software which is then automatically generated. In this case the models might include information about persistent data, processing constraints as well as domain-specific logic. In the words of Fowler, the modelling language becomes the programming

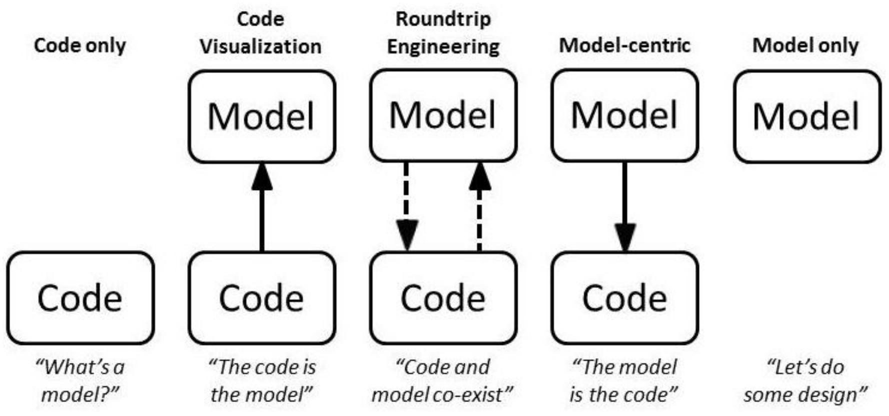


Figure 2: The software modelling spectrum as depicted by Brown [17].

language [44], analogously to how the source code becomes the machine code through code compilation [75].

Model only Finally, in the model only scenario the model is disconnected from its implementation. This is common in large organizations who traditionally are not software companies. After specifying the model it is out-sourced to suppliers and sub-contractors who either develop new software fitting the specification or deliver ready-made off-the-shelf solutions.

The spectrum given by Brown is of course a simplification, a model. Reality is not always that easy to map to the different scenarios and the scenarios can co-exist both as serial and parallel combinations. An example is software development at Volvo Cars. At system-level there is a model which specifies the overall system architecture. From this global model it is possible to generate partial models for specific subsystems. Some subsystems are then realised in-house using a round-trip engineering approach while other subsystems are developed using a code-centric approach where the partial model is seen as a blueprint [44] that the manually derived code must fulfill (a scenario not covered by Brown).

2.1.2 Executable Models

From the perspective of models as programming languages there are three important aspects to consider – the chosen representations of the modelling language, the possibility to deterministically interpret and *execute* a model and the ability to transform a source model to a target representation for a specific purpose.

Representations Over the decades there has been a strive towards raising the abstraction of programming languages [1]. First there were machine languages using 0's and 1's to represent wirings, then came assembly languages and then the third generation languages such as C and Java. For each generation the level of abstraction rose and the productivity of the programmers followed [86]. The aim of using models as programming languages is to continue raising the level of abstraction and thereby also the productivity [73]. By reducing the number of concepts and/or the complexity of the representations the models should be easier to understand than the corresponding representations in C or Java. Exactly which representations that are chosen varies between the modelling languages, just as different third-generation languages have different representations and abstractions. A language where the chosen representations and the expressive power is restricted to a certain application domain is referred to as a Domain-Specific Language, DSL[2] [45, 77].

Executable Semantics An interpreter takes a source program and an input and returns an output [1]. In the case of models as programming languages the source program is a model which the interpreter can execute statement-by-statement in order to validate that the model has the right behaviour and structure [85]. In comparison to programming languages like Java and C, the modelling languages can consist of both graphical and textual constructions which in turn requires a more complex development environment to supply an interpreter that handles both textual and graphical elements as input and output.

Model Transformations In order to define a transformation between two (modelling) languages it is necessary to have a specification for each language. The common way of transforming a model into machine code is to first transform the model into a lower-level program language supported by a code compiler. In this way there is a need for an extra transformation when using a modelling language instead of a programming language such as C or Java [108]. It is also possible to reverse engineer a source into a more abstract target representation [76].

In this view the notion of code in Figure 2 can be either manually derived code or code automatically generated from a model. When the code is automatically generated from the model it is no longer a help in developing code – it becomes the single source of information and serves both as specification and implementation [75].

2.1.3 Model-Driven Approaches

The usage of models as the drivers for software development has many names where Model-Driven Architecture, MDA [17, 68, 75, 79], and Model-Driven Engineering, MDE [15, 65, 95], are among the most popular. The aim of adopting a model-driven approach instead of a code-centric is to handle the complexity of software development by raising the level of complexity and automate tedious or error-prone tasks [80].

MDA defines three abstraction layers and envisions that the development process transcends the layers from the more abstract to the most concrete.

[2]There is a discussion regarding when a DSL is a modelling language or a programming language – see for instance Yu et al. [114] – but we treat them as modelling languages since they from a user perspective abstract away from the low-level implementation details while introducing representations that capture the problem domain [60, 62].

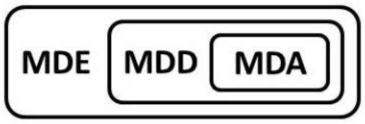


Figure 3: The relationship between MDA, MDD and MDE given as a Venn diagram.

Computational-Independent Models The highest abstraction level in the MDA hierarchy uses the terminology of the domain to define the relevant functional properties. The structural relationship between the terms can also be defined using domain models [70] or taxonomies. It also defines the context – such as intended user groups, interacting systems and the physical environment – of the system while the system itself is treated as a black box, hence the name computational-independent. The computational-independent model is also referred to as a conceptual model [70] and abbreviated as CIM.

Platform-Independent Models At the next level of abstraction the Platform-Independent Models, PIM, introduces computational concepts such as persistence and safety together with interface specifications. The internal behaviour of the system might be defined by using executable modelling elements such as state machines or textual action languages that enhance the graphical elements [73, 85, 109]. The PIM should be possible to reuse across different platforms – combinations of operating systems, programming languages and hardware – by a model transformation that takes the PIM as an input and a specific platform setup as target. In relation to Figure 2 the PIM refers to the model.

Platform-Specific Models A platform-specific model, PSM, represents the lowest abstraction level and details what kind of memory storage should be used, which libraries and frameworks to include and if the system is to be run on one core or a distributed network etc. In relation to Brown's modelling spectrum a PSM is the code.

The concept of executable models is one way of realising the MDA vision since execution allows implementation and validation to be done at the PIM level while the PSM is obtained through automatic transformations [73, 75].

A taxonomy for distinguishing between different modelling approaches is given by Ameller [3] and depicted in Figure 3. Ameller distinguishes MDE from Model-Driven Development, MDD, by seeing MDE as more inclusive than MDD. The reason is that MDD represents a model-centric view and focuses on the transformation of abstract models into more concrete representations. According to Ameller (who cites Hutchinson et al. [60, 58]) MDE acknowledges the importance of a wider range of software engineering factors, including improved communication between stakeholders and quicker response towards changes, besides code generation. The difference between MDD and MDA lies in the fact that MDA only uses the standards conveyed by the

Object Management Group, OMG[3], such as the Unified Modeling Language, UML[4] [2, 12, 44], while MDD is a more pragmatic approach where you use what is most appropriate.

Ameller's distinction between MDA and MDD on the one hand and MDE on the other is consistent with the critique of MDA put forward by Kent [65]. Kent claims that MDA neglects social and organizational issues as well as the need for more differentiated levels of detail and abstraction than CIM, PIM and PSM. Furthermore, different kinds of abstractions are needed depending on the varying qualities of software and that in turn calls for the usage of domain-specific languages instead of generic representations.

From this point of view the distinction between MDE on the one hand and MDA and MDD on the other is that while MDE emphasises the model-centric view it also includes roundtrip engineering with the possibility of reverse engineering concrete representations into more abstract ones. MDA and MDD have a narrower scope and see the transformation from abstract to concrete as the fundamental relationship between model and code.

2.2 Scholarship

According to Boyer there are four functions that form the basis of academia and the roles of those it employs – the scholarships of discovery, teaching, application and integration [13, 14]. The foundation for all four functions is the systematic process of planning, executing, reflecting on and communicating innovation or changes to existing phenomena. Depending on the different functions of the four scholarships the nature of the overall process and the resulting insights will vary accordingly. Throughout the process the interchange with other scholars – such as discussions with peers and publishing – is instrumental to identify, and improve, the impact of the process. The reasoning from Boyer is that embracing the full functionality of the scholarship will remove the artificial conflict between teaching and research and instead practice, theory and teaching will interact and strengthen each other.

Figure 4 gives our interpretation of how the four functions represent different aspects of the scholarly process. For brevity we refer to each scholarship by its name where the initial letter has been capitalised, e.g Discovery refers to the scholarship of discovery.

Discovery The pursuit of new discoveries contributes in general to our knowledge and understanding of the world - more specifically discoveries are critical for a vital and intellectual climate within academia. Boyer stresses that not only is the discovery as such important, the systematic and disciplined process of conducting original research should also be stressed. The primary audience for new discoveries are academics within the same discipline while students and practitioners are also possible recipients and/or benefactors.

Teaching Just as for any of the other academic functions, teaching can and should be done in a systematic and disciplined way. As the terms scholarship of teaching

[3]`http://www.omg.org/` – Accessed 14 February 2014.
[4]`http://www.uml.org/` – Accessed 14 February 2014.

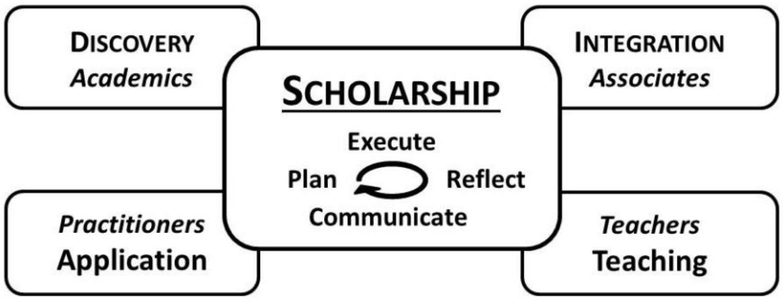


Figure 4: The four functions of SCHOLARSHIP with respective key *Scholars*.

suggests, Boyer expects the outcome to be communicated to other teachers first of all but the insights should serve the interests of the students either as shared theoretical insights or by the application of new insights on the teaching activities related to the discipline. The aim of teaching as a scholarship is to find new ways of encouraging active learning and critical thinking while providing a foundation for life-long learning. In this sense scholars are also learners where the interaction with the students is an opportunity to improve one's own understanding of the subject as well as the process of teaching and learning [57].

Application The scholarship of application refers to applying established discoveries to problems outside of academia. But it can also work in the opposite direction where the direction of future discovery is influenced by the challenges encountered in non-academic settings. In this way theory and practice interact and feed each other with new insights that will then be communicated to the practitioners (commercial or non-profit organizations, individuals or governmental bodies etc.) but also to academics within the discipline.

Integration The scholarship of integration seeks to illuminate established facts and original research from new perspectives or interpret them in new ways. As a consequence it opens up new disciplines or – using the terminology of Polanyi [83] – identifies over-laps among disciplines. Since integration is inter-disciplinary or even spans multiple scholarships, the assumption that the intended audience are knowledgable on all of the included topics becomes moot. Rather, the recipients will be specialists within some of the disciplines included in the integration, but not knowledgeable in all. This is a challenge not only in how to define a systematic methodology but also in how to communicate the results to reach across to all concerned parties[5]. Boyer also refers to integration as synergy or synthesis since it establishes connections among different disciplines or scholarships.

Boyer explicitly states that there is an overlap among the four and insights can fall into one or more scholarships while the intended audience can include more than one

[5]This thesis is an example of the challenges involved in communicating the research outcomes by integrating language technology, software engineering and engineering education research while applying MDE to disparate domains such as telecommunications and automotive software development.

kind of scholar [14]. An explicit example is how Teaching has elements of applying new discoveries from education and the subject area in a classroom context [113] or implicit examples such as knowledge transfer – applying academic discoveries in new organisations [40].

3 Research Methodologies

The included publications have been conducted using three different research methodologies – action research [64, 71], case studies [92, 121] and design science research [55, 103]. The research methodologies are not mutually exclusive, in fact they share the basic steps to *plan, execute, reflect* and *communicate*. The methodologies can also be combined – i.e. a case study can be applied within a design science research method to demonstrate the applicability of the design [82].

3.1 Action research

Action research is used to understand, but most of all improve, real-life situations in an iterative way [8, 47]. In relation to software engineering action research has been used mostly for implementing organizational changes related to software development – such as process improvement [61] and technology transfer [49] – in contrast to developing software artifacts [93]. Action research has also been advocated as a suitable research methodology for educational purposes. For instance, Stenhouse argues that action research can contribute both to the practice and theory of education and that communicating the resulting insights to other teachers is important in order to promote reflection within the community [111] (cited in [31]).

Action research is conducted over a series of iterations where each iteration can be broken down into three steps [36]:

Planning Defining the goals of the action, setting up the organization that will carry out the change and acquire the necessary permissions, knowledge and skills.

Acting Executing the plan and collecting the data that reflects the change. The data can consist of interviews, surveys, logs and/or observations but also documentation of the decisions that are made and their rationale.

Reflecting Evaluating the impact of the change in terms of collected data, discussing the organization and process with the participants, analysing the collected data as well as *communicating* the results to the relevant audiences.

A new iteration is initiated as new actions are identified during reflection. The detailed content and duration of each step varies depending on the objectives of the iteration and how the context has changed due to earlier interventions.

3.2 Case study

A case study aims to analyse one phenomenon or concept in its real-life context which in turn means that it is difficult to define the border between phenomenon and context

[10, 91, 121]. A question that is still under debate is if the findings from a case study can be generalised beyond the scope of the study or not, see e.g. [32, 43, 66].

According to Runeson et al. a case study can be broken down into five major steps which they claim can be used for all kinds of empirical studies [91]:

Case study design The design should specify the aim and the purpose of the study as well as lay out the overall plan.

Preparation for data collection The procedures and protocols for collecting the data relating to the objectives are defined.

Collecting evidence Depending on the design of the study the data can be collected from primary sources such as interviews and observations, indirect sources like monitoring tools or video recorded meetings but also data that already exists – e.g. requirements or source code – can be collected.

Analysis of collected data If the aim of the study is to explore a phenomena in its context the collected data is analysed inductively [91] – coding the data and then forming theories and hypotheses from the identified patterns among the codes. On the other hand the analysis can be deductive by its nature [91] and aim to validate or refute existing theories. In this case the collected data is compared to the predictions of the theories to be validated. Seaman argues that the analysis of the data should be carried out in parallel to the data collection since analysis can reveal the need for new data and allow the collection of data to confirm or refute emerging results [98].

Reporting The outcome of the study is reported to funders, industrial partners, educational communities etc. Different reports can be necessary depending on the intended audience.

Robson [89] states that case studies can be used to *explore, describe* or *explain* a phenomena in its context but case studies can also be used in an *emancipatory* way by changing aspects of the phenomena to empower certain stakeholders. This is in contrast to Runeson and Höst who argue that a case study is purely observational and that empirical studies that aim to change an existing situation should be referred to as action research [92].

In the case of *inductive* studies – where the aim is to generate new hypothesis and theories with respect to the investigated phenomena [91] – Seaman states that ideally further data collection can help to refute, validate or enrich emerging theories but this is not always possible [98]. Furthermore, Flyvbjerg argues that in the case of generating new insights it is more interesting to focus on the exceptional data than data commonly found throughout the study since this opens for new insights regarding the phenomena and re-interpretations of established discoveries [43].

3.3 Design Science Research

Design science research is a research methodology for creating software artifacts – designs – addressing problems encountered in research or practice [55, 82, 103]. In this context Simon defines a design as a human-made or artificial object [103] and the aim of design science research is the systematic investigation of the design of computer systems used for computations, communication or information processing [56]. While different designs are driven by different needs – such as addressing problems raised in previous research or by practitioners – the outcome should not only be a new artifact but also a theoretical contribution to the relevant field of research [82].

According to Peffers et al. the design process can be described in six steps [82]:

1. Identify and motivate the problem and relate it to state-of-the art as formulated by previous research.

2. Define the objectives of the artifact that answers to the problem above. The objectives can be quantitative – such as runtime metrics or figures for precision and recall, or qualitative – like a description on how the artifact supports new solutions.

3. Design and develop the artifact so that it delivers a research contribution either in how it was implemented or in what it accomplishes.

4. Demonstrate the artifact on an instance of the original problem. Depending on the objectives the demonstration can be anything from a case study to a proof-of-concept implementation.

5. Evaluate the artifact in relation to the objectives and state-of-the-art. Iterate steps 3-5 until the artifact meets the objectives.

6. Communicate the outcome of the design, both in terms of the artifact as such but also the insights gained throughout the process.

Where action research, case studies and grounded theory emphasise the investigation of a phenomena in its context, design science research emphasises the development of an artifact. Thus the latter recognises the possibility to investigate designs outside of their original contexts [117].

4 Contributions

The included publications are described per scholarship, for each scholarship introducing the shared context of the publications and then the individual contributions of each publication. The first scholarship is Teaching since this is the origin for our investigations into software models and MDE. Discovery is next as it builds on the modelling technologies introduced for Teaching to explore the possibilities of natural language generation from software models. Then comes Application as we wanted to validate if our understanding of MDE had bearing in industry. Finally, insights from the three scholarships are pooled together in Integration where a project involving three large companies has a central role.

4.1 Scholarship of Teaching

The scholarship of Teaching is represented by two publications on improving how to teach MDE. The first publication, *Executable and Translatable UML – How Difficult Can it Be?* [27], is a case study on the introduction of executable models while the second publication, *Pair Lecturing to Model Modelling and Encourage Active Learning* [23], reports on the impact of pair lecturing MDE. The two papers are related in that they both address the challenge of transferring an understanding of the problem into a validated solution.

4.1.1 Mastering Modelling

In software development the process of understanding a problem and defining a solution using relevant abstractions involves an implicit form of modelling [46]. During this process the developer has as an internal model of the system which can be shared and discussed with others. Over time the developer's internal model evolves through the interactions with other stakeholders, the changing requirements and context of the system as well as through the validation of the implementation.

Cognitive Apprenticeship Mastering the process from understanding the problem to validating a solution is not necessarily an easy task. From the perspective of MDE education it is not obvious for all students how to share and discuss different solutions – and even harder to come up with them in the first place.

Cognitive apprenticeship [18] is an approach to instructional design that aims to teach how masters form their internal model and then step-by-step formalise it into a solution that fulfills the requirements. In this way the tacit knowledge as well as the alternative routes are made explicit so that novices – apprentices – learn from imitating and reflecting on the practice of a skilled master. The learning possibilities are structured as six teaching methods:

Modelling The master verbalises her own cognitive process of performing a task – including dead-ends – so that the novices can build their own cognitive model of the task.

Coaching The master observes the novice performing a task and offers feedback and hints in order to further develop the novices' ability.

Scaffolding Compared to scaffolding the novices get less support while performing a task in a setting which is assessed by the master to fit the abilities of the novice.

Articulation The novices are given the opportunity to articulate knowledge, reasoning skills as well as problem-solving strategies either in collaboration with another novices or in interaction with the master.

Reflection The novices compare their ability to that of other novices or the master but also to the emerging internal cognitive model of expertise.

Exploration The novice is given room to solve problems without the help of the master.

As teachers we need to facilitate a learning environment where students can explore MDE concepts in different contexts and validate the completeness of their models. These are challenges we have encountered in our own course on MDE.

Teaching Model-Driven Engineering The course runs over eight weeks and corresponds to 200 hours. Most of the time the students spend on a course project working on an open-ended problem. The first part of the project consists of developing blueprint CIMs of a hotel reservation system and then in the second part developing an executable PIM. The project work is supported by two lectures per week except the last week when the students demonstrate their projects. A majority of 90-130 students are at their third year in one of three different bachelor programs while some come from a master program in software engineering. During the evolution of the course we have aimed to address both how to capture domain knowledge using models and how to reason about software models as well as how to validate the functionality and structure of the models.

Cognitive apprenticeship was not originally part of the motivation of the first included paper [27] – the connection was made while synthesising the thesis. For the second included paper [23] the link was made during our reflection on the outcome while preparing for communicating the outcome to a broader audience.

4.1.2 *Paper 1:* How Difficult Can it Be?

The publication presents the outcome of letting our students develop executable PIMs as part of their project work [27]. Before we introduced executable models into our course the students spent 175 hours preparing models in our course and then 200 hours in the following course to manually transform them into Java. As a result it took months before they were able to test their design decisions. The models and the code rarely correlated and the different models were inconsistent with each other. And since the only way to validate the models was by manual model inspection the students often had a different view than the teachers on the level of detail in the model.

From an MDA perspective an executable PIM would be the remedy since the interpreter would give the students constant feedback [73, 85] as they explore how to transform their CIM into an executable PIM. Using the terminology of cognitive apprenticeship the supervision time was used for modelling and coaching while scaffolding and exploration would distinguish the work the teams did on their own. Hopefully the students would spend time on reflection on their own initiative but the demonstration at the end of the course was an opportunity for us as teachers to ensure they did reflect on their own ability.

The question we wanted to explore ourselves was to which extent it would be feasible to develop an executable PIM as part of the course?

During the first part of the project the teams consisted of eight students that split into two teams of four to develop executable models. Each team had a total of 300 hours to their disposal and we defined a set of evaluation criteria to specify the expected design and functionality. In the first year, 2009, there were 22 teams and in 2010 the number of students had increased and 28 teams of four participated. From the evaluation in 2009 we learnt that the students wanted more tool supervision so in 2010 one of the earlier students spent 22 hours on helping out with tool related issues.

In 2009 all 22 teams managed to deliver an executable model within the time frame but four teams failed to meet the evaluation criteria. In 2010, 25 out of 28 teams delivered on time as well as following the criteria but two teams failed to deliver on time and one team did not meet the criteria. The 25 teams that succeeded to meet both time constraints and evaluation criteria had in general more elaborate models than the 18 teams of 2009. In 2010 we introduced a survey to get a more detailed picture of how much time the students had spent learning the tool and its modelling paradigm. Out of 108 students 90 responded and 75 of those estimated that they required up 40 hours before they were confident using the tool. Our own evaluation of the models gave that the quality in terms of details and consistency had improved in general and in some cases went beyond what we thought possible, given the context.

4.1.3 *Paper 2:* Pair Lecturing to Model Modelling

While the introduction of executable models changed the students project work to the better we still found that the students struggled to apply the lecture content to their project. Lectures tend to present a neat solution to a problem and in the case the process for obtaining the solution is given that is also given as a straight-forward process [112]. What we wanted to do was to increase the interaction with the students in order to adjust the lecture content to their needs since students learn more when they are actively involved during lectures [51, 84].

Pair lecturing lets us do just this. In the context of cognitive apprenticeship we used the lectures to model, articulate, reflect and explore different ways of modelling the domain of course registration. In this way we make explicit our own individual cognitive processes in interaction with the students where we let them be the masters of the domain and we are the masters of MDE. Together we explore different ways of modelling the same phenomena while discussing their pros and cons. The lectures become an arena for how to use models to organise the functionality and structure of the domain and how information from one kind of model can be passed on to other kinds of models and enriched with more details from the domain.

The publication aimed to answer two questions *Can pair lecturing encourage students to take a deep approach to learning in lectures?* and *What are the pros and cons of pair lecturing for students and teachers?* In order to answer the questions a survey was used to collect the impressions of the students – as assessments on a five-grade Likert scale and as comments in relation to four statements.

From the survey it was possible to conclude that pair lecturing enabled students to be more active during the lectures than what they were used to. One obvious way of for a student to be more active is to participate in the on-going discussion. However, in a lecture hall with 100 students it is difficult to get everyone involved in the discussion – both practically but also since not everyone wants to state their opinion in public. But by posing two solutions to one problem even the silent get more active since they need to chose one of the solutions to accommodate to their project or come up with their own. We also found that the students felt they could influence the lecture content more than usual which gave them more opportunities to address the issues of MDE they found most challenging. The students also reported on remembering more after the lectures.

Regarding the pros and cons the students found that too many opinions could be confusing while they appreciated that complicated concepts were explained twice and in different ways by the two teachers. As teachers we found that we were out of comfort zone since we could not predict where the student interaction would take us. The most important change for us was the new possibility for *reflection-in-action* [96].

4.2 Scholarship of Discovery

This section will first relate the concept of natural language generation to model transformations before describing the contributions of the included publications – *Natural Language Generation from Class Diagrams* [21] and *Translating Platform-Independent Code into Natural Language Texts* [22]. While originally published as exploratory case studies it would have been more sound to label the investigations as design science research due to the fact that it is the transformations – the designs – that are in focus while the designs are motivated by the findings of Arlow et al. where the original context has been abstracted away. The validation of the technique is limited to the feasibility of generating natural language descriptions from software models – there is no validation of the usability of the texts in an industrial context.

4.2.1 Model Transformations

A model transformation is a set of rules that define how one or more constructs in a source language are to be mapped into one or more constructs of a target language [68] together with an algorithm that specifies how the rules are applied [75]. The rules are specified according to the syntactical specification of the source and target languages and since the specification is a model of the modelling language it is referred to as a meta-model [5, 67]. A translation is transformation where the source and target languages are defined by different meta-models [118].

Depending on the levels of abstractions of the source and target languages a translation has one of the following characteristics [76]:

Synthesis The translation from a more abstract source to a more concrete target language is referred to as a synthesis translation. An example is the translation from source code to machine code where the compiler synthesises the information in the source code with information about operating system and hardware [1].

Reverse Engineering To reverse engineer is the opposite of synthesising – the translation removes information present in the source when defining the target [30]. Brown's notion of code visualisation [17] is an example of reverse engineering.

Migration The source is migrated to a target when the languages have the same level of abstraction but different meta-models, e.g. when porting a program from one programming language to another [76].

Natural Language Generation from Software Models The publications realise the transformation from model to natural language in the same way, using a two-step process. The first step is to transform the relevant parts of the model into the equivalent constructs as defined by a linguistic model – in other words a grammar. The linguistic model is then used to generate the natural language text. The two transformations relate to Natural Language Generation which transforms computer-based representations of data into natural language text in a three-step process – text planning, sentence planning and linguistic realisation [9, 88]:

Text planning During text planning the data to be presented is selected and structured into the desired presentation order.

Sentence planning Then the individual words of the sentences are chosen and given their sequential order according to the syntactical structure of the target language.

Linguistic realisation Finally, each word is given the appropriate word form depending on case, tense and orthography.

Depending on the purpose of the target language, the resulting translation can either be an example of synthesis, reverse engineering or migration. I.e. a textual summary of the model would be a reverse engineering transformation, a migration translation would yield a text that exactly paraphrases the model and a translation that not only paraphrases the model but also adds contextual information would be a synthesis transformation.

Literate Modelling Both the included publications under Discovery stem from the same problem reported by Arlow et al. [4]. From their experiences of the aviation industry they claim that models are not always suitable for organising requirements into source code. Their example is based on the concept of code-sharing (one flight having more than one flight number) worth millions of pounds every year in revenue for the airlines. This is denoted by an asterisk (*) when transformed into a class diagram. They go on to claim that in a class diagram all requirements looks the same – making it impossible to distinguish which requirements are more important than others.

Arlow et al. further claim that in order to validate the correctness of a model it is necessary to understand the modelling paradigm (object-oriented in their case), knowledge of the used models to decipher the meaning of the different boxes and arrows as well as acquiring the necessary skills to use the different modelling tools that are used for producing and consuming the models [4].

The aim of the two publications is then to explore the possibilities to paraphrase the models as natural language texts – a medium stakeholders know how to consume [42]. The idea is that since all changes to a system are done at the PIM level the generated texts residing at the CIM level will be in sync with the generated PSM. Thus documentation and implementation are aligned with each other and there is a single source of information for both documentation and implementation.

4.2.2 *Paper 3:* Language Generation from Class Diagrams

For the first translation the input was class diagrams while the output paraphrased both the included classes and associations in a textual format. Any comments embedded in the diagram were taken at face value and appended to the corresponding text element. In this way the underlying motivations of the diagram is carried over to the textual description.

Our transformation rules were inspired by the work done by Meziane et al. [78]. They generate natural language descriptions of class diagrams using WordNet [41], a wide-coverage linguistic resource which makes it useful for general applications but can limit the use for domain-specific tasks. In contrast our transformation reuses the terminology of the model to generate a domain-specific lexicon. Since the transformation rules are defined using the meta-model of the modelling language they are generic for all models that conform to the meta-model. In combination these two qualities enable the described approach for transformation from model to natural language text to be applied to any domain.

4.2.3 *Paper 4:* Translating Platform-Independent Code

Where the prior publication translated a class diagram that defines the structure of the software this publication takes an action language – a behavioural description of the model – as its input. The difference between an action language and a programming language compares to the difference between a programming language and assembly code – where the programming language abstracts away from the *hardware* platform the action language abstracts away from the *software* platform [75]. This means that when a programming language enables computations without knowing the number of registers or how the structure of the stack, an action language enables the addition of the ten first values of a set without specifying it as an iteration of a list or a increment over an array. As a consequence, according to Mellor et al. [75], just as the computations are reusable across hardware, the actions are reusable across software.

To our knowledge this was the first attempt of generating natural language from an action language. In comparison to previous work on generating natural language from code (e.g. see [105, 106, 87, 50]) our approach enables one transformation to be reused across many software platforms. This means that the generated texts can be reused independent if the behaviour is realised using C, Java or Python. Another benefit is that if the transformation is done at code-level the transformation has to sieve away the platform-specific details [105] as well as be re-implemented for each software platform.

4.3 Scholarship of Application

The contributions under Application consist of two publications that report on the possibilities – *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* [26] – and challenges – *Limits of Model Transformations for Embedded Software* [54] – of multi-paradigmatic modelling from a case study in industry.

4.3.1 Multi-Paradigmatic Modelling

The case study at Ericsson was a testbed for evaluating the possibilities of combining multiple domain-specific modelling languages for implementing the channel estimation of a 4G base station.

Domain-Specific Modelling In relation to modelling techniques being applied to more complex systems consisting of domains with different characteristics, the necessity to apply modelling languages with appropriate representations has risen as well. A platform-independent modelling language tailored for a specific domain is referred to as a domain-specific modelling language, DSML [63].

In this context a domain represents one subject matter with a set of well-defined concepts and characteristics [101] that cooperate to fulfill the interactions of the domain [85]. This definition of a domain is in line with what Giese *et al.* [48] call a horizontal decomposition and ensures the separation of concerns between the domains [37] as well as information hiding [81]. Furthermore, each domain can be realised as one or more software components as long as these are described by the same platform-independent modelling language [85].

The aim of a DSML is to bridge the gap between how domain concepts are represented and how they are implemented. Therefor a DSML consists of a set of representations relevant for the domain and one or more code generators that transform a program expressed by the DSML into code for a specific platform. DSML is another way to approach the challenges of validating models as expressed by Arlow et al. – instead of knowing how to map the modelling paradigm and the modelling concepts to the domain [4], the modelling languages uses a paradigm and representations relevant for the domain.

Multi-paradigmatic modelling is a research field that addresses the challenges that arise when using multiple DSMLs within the same system. Among the challenges are co-execution, integration of languages and transformation from multiple sources to a shared platform [52].

Channel Estimation for 4G The underlying case study for the two publications was a testbed for using two different DSMLs for implementing the channel estimation of the LTE-A uplink of a 4G telecommunication system at Ericsson AB. The system was demonstrated at the Mobile World Congress in Barcelona 2011. The requirements included unconditional real-time constraints for calculations on synchronous data and contextual determination of when signals should be sent and processed.

The system was analysed as consisting of two domains with their own characteristics and representations. The first domain was the signal processing domain which is signified by more or less fixed algorithms that filter, convert or otherwise calculate on incoming data while keeping a minimum interaction with external factors. The second domain was referred to as the control domain since it controlled the flow of execution and was responsible for interacting with the surrounding environment, determining which signals to send and process given that the context changed every millisecond.

Traditionally such a system would be realised using C, at least at Ericsson. The problem is that the result is a mix-up of language-dependent details, hardware optimisations and desired functionality. Instead a multi-paradigmatic modelling approach was tried out were the signal processing domain was implemented in a textual language without side effects that was embedded in functional language [6, 7]. The control domain was implemented in the same object-oriented modelling language that we had previously introduced in our course [73, 85].

The case study set out to answer four research questions:

1. To which extent is it possible to develop the signal processing and the flow of execution independently of each other?
2. How can the sub-solutions be integrated into one application running on the designated hardware?
3. Which are the key challenges to consider when generating imperative code from a functional modelling language?
4. To what extent is it possible to reuse existing model transformations for new hardware?

Questions 1 and 2 were answered in *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* while questions 3 and 4 were addressed in *Limits of Model Transformations for Embedded Software.*

4.3.2 *Paper 5:* Multi-Paradigmatic Modelling of Signal Processing

The development process relied on the possibility to validate the domains independently. For this to be possible it is important that the modelling languages are executable in their own right and that they later on can be translated into deployable code, after the models have been validated to have the right structure and behaviour. A benefit of the chosen approach was that we avoided the difficulties regarding tool and language integration recently reported on by Mohagheghi et al. [80] and Torchiano et al. [116].

Since the DSML used for the digital signal processing was embedded in a functional language the domain was independently developed and validated using the interpreter of the functional language. The control domain was implemented and validated using the interpreter that came with the executable modelling tool. In both cases the interaction with the other domain was stubbed, using dummy values for testing the signal processing and simulating the execution of signals in the control flow. Regarding the DSMLs appropriateness for the domains, the experts in respective domain stated that the chosen representations of respective language were not ideal since it was a challenge to map the domain concepts to the representations of the DSMLs.

The interfaces between the domains could be negotiated through Scrum meetings [11, 97] since the two teams were located in the same office landscape. Thus the interfaces could be detailed as the understanding of the domains grew during implementation instead of in advance. The implementation of the interface was then hand-coded since it required multi-core deployment which neither of the DSMLs had representations to handle. Full interaction of the two domains could not be done until after code generation.

4.3.3 *Paper 6:* Limits of Model Transformations

In the case of transforming a functional DSML to an imperative platform we encountered a problem that could not be solved under the time constraints. During the transformation new variables were introduced for intermediate results. Since the available memory on the chip was scarce all variables were limited to 16 bits of memory. However, in some cases the intermediate results needed 32 bits in order for the calculations to be precise enough. If all intermediate variables had been assigned 32 bits the generated code would not fit on the chip. Since the intermediate variables were introduced by the transformation they could not be marked in the models for special treatment during transformation [75]. In the end the only solution feasible within the time frame was to manually edit these variables in the generated code. If Multi-Paradigmatic Modelling is going to be successful it is necessary to pay more attention towards cross-paradigmatic transformations when the target platform comes with hard constraints on memory or processing capabilities.

In the case of the object-oriented DSML there already existed a model transformation proven to be efficient [102]. However, the transformation needed to be updated to accommodate for the new platform. The transformation expert did a first iteration and the results seemed promising when the decision was taken further up in the organisation that his services were needed better elsewhere. In the decision between finishing the update of the transformation themselves or to manually edit the generated code the engineers decided that it would be quicker to post-edit the generated code. From an organisational point-of-view the introduction of MDE can imply that new skills are needed. If the skills are scarce among the engineers the experts can quickly become bottlenecks in the development.

4.4 Scholarship of Integration

Integration consists of four publications with a shared origin in a study of MDE that we undertook at three companies. The publications are presented in the same order as they were published since the three first publications present more specific aspects of the data while the fourth presents a fuller analysis – which explains why it was finished last. Since the collected data was used and analysed in different ways for the four publications the analysis procedure and results will be presented under each publication.

First out is a publication in relation to *Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem?* [119]. The paper integrates data from an earlier study by Hutchinson et al. [58, 59, 60, 120] with a subset of the interviews from our own MDE study in order to explore the impact of tooling on the adoption of MDE.

The second paper is *Extending Agile Practices in Automotive MDE* [39]. At the same time as our own study an internal and independent study was conducted at one of the involved companies. The included paper is the result from integrating the two studies in relation to the possibilities and challenges concerning agile development from an MDE perspective.

For the third paper we integrated the text generation techniques developed under the scholarship of discovery with interview data from our MDE study to came up with *Enabling Interface Validation through Text Generation* [24].

Finally, the fourth paper shares the general analysis from our MDE study as *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* [28]. The reported integration is a validation of Hutchinson et al.'s aforementioned research on industrial adoption of MDE, together with new insights specific to our own study.

4.4.1 A Study of MDE at Three Companies

The study was planned during the autumn of 2012 and then executed in the spring of 2013 with the final analysis and dissemination taking part during the autumn of 2013. There was a considerable overlap between execution and analysis which enabled new data to be collected during the analysis in order to follow up on emerging themes and validate spiring hypothesis [98]. The study was an international collaboration with Jon Whittle at Lancaster University. He was involved in an earlier study assessing the industrial application of MDE which included a survey with more than 400 respondents and 19 interviews at 18 companies representing 10 different industrial sectors [58, 59, 60, 120].

Motivation The aim of our study was to compliment the earlier study – which we refer to as Hutchinson et al. since John Hutchinson was the primary investigator – with a deeper and more narrow investigation. In this context we wanted to answer two questions, *What are the similarities and differences in the way that the three companies have adopted and applied MDE?* and *How does the nature of practice at the three companies agree or disagree with the findings from Hutchinson et al.?*

Context The study involved three companies; Ericsson AB, Volvo Cars Corporation and the Volvo Group. The three companies had in common that they were all in a transition into more agile software development while their experiences of MDE varied.

At Ericsson we collaborated with a unit developing software for radio-base stations. The unit has been involved in various MDE projects since the late 1980's and have primarily used UML for both documentation, specification and code generation in relation to their software development. We interviewed nine engineers, ranging from newly employed software developers, senior architects to managers. The interviews were conducted from January to March 2013.

Electronic Propulsion Systems is a new unit at Volvo Cars that develop software for electric and hybrid vehicles which started to use MDE in 2010. The interviewees were 12 all-in-all, mainly experts in physics or electric engineering encoding their knowledge in a modelling language that supports code generation. Just as for Ericsson the interviews took place during January to March 2013.

The Volvo Group is a code-driven organisation but there are pockets of MDE development, mainly at the top-level system planning with partial code generation. Due to the fact that the Volvo Group entered the project during the spring of 2013 and that there were fewer MDE practitioners we only interviewed four engineers. The interviewees represented Truck Technologies and had both managerial roles as well as being developers or architects. The later entry date meant that the interviews were conducted between March and June 2013.

Data Collection The interviews were semi-structured and lasted for approximately 60 minutes each. A typical interview started off with general questions about personal experiences and attitude towards MDE as well as educational background and current role in the company. Depending on the outcome of the first questions the rest of the interview would go into more depth, exploring topics that struck the interviewer as most interesting or controversial. The interviews were audio recorded and then transcribed. The interviews were complemented with additional data collection in order to cross-verify the quality [91]. Besides observations we also used informal interactions at the coffee machine or lunch table since this allows for more spontaneous interaction and data collection from other practitioners than those interviewed [34]. Our findings were then presented at technical meetings to the engineers as well as at regular process meetings with company representatives where the findings were discussed and we received additional interpretations of the data. During the complimentary data collection field notes were used to document insights and impressions.

4.4.2 *Paper 7:* Are the Tools Really the Problem?

The immaturity of the tools is an often mentioned reason for why MDE adaption fails [69, 99, 116, 115]. But we also know from Kent [65] and the study by Hutchinson et al. that social and organisational aspects impact the adoption of MDE in industry [58, 60]. The questions we investigated in this publication was *To what extent poor tools hold back adoption of MDE?* and *What aspects – both organisational and technical – should be considered in the next generation of MDE tools?*

The taxonomy was induced from the interviews conducted by Hutchinson et al. [58, 60]. Each transcript was coded by two researchers at Lancaster University. The codes were then grouped into four themes – technical, external organisational, internal organisational and social factors – each factor with its own categories. We then used the data collected at Volvo Cars and Ericsson to validate the taxonomy by mapping tool related issues in the interviews to the taxonomy. Whenever there was any exceptional issue or it was unclear how to classify an issue this was noted down and discussed among the researchers.

The technical factors relate to issues such as specific functionality like code generation, the impact on quality and productivity as well as complexity of the tools and the modelling languages. Adapting tools to governmental or commercial standards or the cost of buying tools are examples of external organisational factors. Under internal organisational factors we grouped the need for new skills when adopting MDE, finding the right problem to start with and how to adapt the existing process with the possibilities enabled by the tools. Social factors identify among other things how engineers relate to the tools due to the (lack of) trust towards tool vendors and tools. The four factors are to a certain extent overlapping and it is possible to arrange the identified categories in other ways. We also expect new aspects to be identified as new issues are found in future studies.

The taxonomy can be used for evaluating existing tools or as a checklist when developing new tools but we also identify a number of aspects that warrant more attention from the MDE research community. We therefore propose that:

- tools should match the intended users instead of forcing people to adapt to the tools;
- while commercial tool vendors focus on code generation from low-level designs the research community could explore tools for creativity and high-level designs;
- more effort is needed in identifying which problems are suitable for MDE;
- researchers pay more attention to understand how processes and MDE tools can improve overall software development and
- open MDE communities are important since practitioners often feel isolated due to the lack of on-line forums where tool related issues can be discussed.

The last proposition was identified during the validation phase and an appropriate category should be included under social factors. We also found that a single tool issue often maps to more than one category in the taxonomy which shows how the factors interact in determining the success of MDE adoption.

Returning to the original questions of investigation we found that the technical aspects of the tools do not support those who try to adopt MDE but also that the organisational and social factors are just as important to consider.

4.4.3 *Paper 8:* Agile Practices in Automotive MDE

The automotive industry is traditionally oriented towards mechanical and electrical engineering using a rigorous process for planning, developing, integrating and finally testing the integrated software, hardware and mechanical parts in a prototype vehicle before going into mass production. Each step of the process is given firm start and end dates minimising the possible overlap between activities.

During the last twenty years the influence of software has grown exponentially so that a modern car contains 100 or more individual hardware units [19, 38]. The rigorous process used for mechanical and hardware development is a barrier for software development since each iteration lasts for approximately half a year with the consequence that software designs specified during the planning stage cannot be validated until several months later. One problematic area for software development is the interfaces between the top-level components since these are frozen at the planning stage. If the need for new signals or parameters is identified during implementation these changes cannot be included in the interface until next iteration.

At Volvo Cars software development teams that work within a top-level component have been successful in applying Simulink[6] for developing software for electronic

[6] A modelling language commonly used for control engineering tasks such as regulating water levels or state transitions of a system: http://www.mathworks.se/products/simulink/ – Accessed 14 February 2014.

propulsion – even though the team members have no training in software development. Instead the engineers have a background in electrical engineering or physics. An important aspect here is that the engineers have used Simulink, and the related MatLab[7] tool, during their university training. The resulting models are then transformed into code by a code generator. By building up a simulation environment it is also possible to validate the behaviour of the sub-system models before the software is integrated on a prototype vehicle. This allows for much shorter iterations between changes and shortcuts the necessity of specifying the software before development since the specifiers implement their own domain expertise.

Our publication sets out to answer the question *Which are the challenges and possibilities for a more agile software development process on a system level?*. And the way we did it was through two independent case studies.

The first study was launched internally by Cars in order to explore how the current process could become more agile. The second study was our own on MDE at three different companies. The first study used semi-structured interviews as the main source of data, interviewing eight software developers between May 2012 and April 2013. In the publication we narrow the number of interviewees at Cars in the second study to eight so that the two sets of informants are consistent in terms of background and responsibilities. Though the two interview sets were analysed independently of each other they came to the same conclusion – something that was recognised by one of the managers involved in facilitating both studies.

The identified challenge towards more agile development at system level was the freezing of the interfaces. The engineers reported on different work-arounds to overcome the problems such as defining extra signals or parameters that might not be used. As a result the interfaces are difficult to understand with regards to the actual behaviour and consume more memory and processing capabilities than necessary. These problems are dealt with but lead to more work later in the development process than originally planned for.

To identify possible solutions to the identified challenge a system architect with responsibility for the system-wide interfaces was interviewed. The perceived remedy was more MDE. The focus in the future should be on the teams developing the software and not on the planning stage in the development cycle. By extending the current tool that manages the interfaces to also execute consistency checks the teams could add new signals as they were needed and old signals that were never used could be discarded. The consistency check would ensure that the new signals were compliant to legacy signals in a manner similar to what Rumpe advocates [90].

4.4.4 *Paper 9:* Validation through Text Generation

As seen in the previous publication the interfaces are problematic when they are overloaded with unused signals – it becomes difficult to know which signals are used and if there is an intended order between the signals. The problem of understanding the interface implementation was not unique to Volvo Cars but encountered at the other companies as well. It seemed to be an opportunity to try to integrate the insights regarding model transformations from Discovery with a problem found in industry.

[7]http://www.mathworks.se/products/matlab/ – Accessed 14 February 2014.

The aim of the design was to explore if a text describing an interface can be generated from an existing test model. The reason for using a test model is that it should capture the desired functionality of the system behind the interface, describing in which sequence the signals are supposed to be sent and what the expected response is [53]. The problem is that it is a model and not necessarily understood by all stakeholders, hence a possibility for natural language generation.

The contribution consists of generic transformation rules that transform a test model into a natural language text. The text specifies the intended order between the signals and also filters out the signals that are not covered by the test model. In contrast to the earlier publications under Discovery this transformation does not rely on an intermediate grammar for linguistic realisation. Instead the concepts of the source language are mapped into holes of a template representing the target language. In this way the transformations are easier to manipulate since they do not require an understanding of the target language's linguistic paradigm. And as we saw under Application it is positive if we can keep the complexity of the transformations low.

While reported as a case study the publication is more appropriately defined as a design study since we due to time constraints could not evaluate the outcome in its original context. The lack of validation in the original context means that the question of what we will find when balancing the complexity of the transformation with the fluency of the generated text is still open. The same is also true for how useful and readable the generated texts are and what measures can be taken to improve them.

4.4.5 *Paper 10:* Model-Driven Engineering at Three Companies

For this publication the focus was two-fold – to induce new hypothesis from the data [98, 43] and validate the findings of Hutchinson et. al [58, 59, 60, 120]. The interviews were transcribed and each transcript was coded by at least two researchers. The codes were then grouped into themes where the most interesting themes were selected for publication. Each selected theme was validated both by triangulation of data and by letting company representatives confirm the outcome before we submitted the paper. After this analysis was finished we extracted the findings from Hutchinson et al. and for each finding we went through the transcripts to see if the finding was supported or refuted by our data.

The first reported insight is that MDE enables software development to be brought back in-house. This is important to shorten lead times between the decision to implement a feature and its actual integration on platform. A key factor here is that it also enables a more agile way of working since software development can be done by the domain experts that traditionally specify what suppliers and consultants are supposed to deliver. In this way the dependency on the knowledge of external actors is mitigated as well as the risk that they misinterpret the specification or do not deliver on time. It also enables the development of new features in an exploratory fashion which is not possible if the functionality has to be detailed before development. The change is possible since the domain experts can encode their knowledge in a suitable domain-relevant language that supports the specification to be transformed into code.

Hutchinson et al. flag for the possibility that the gain of adopting MDE can be counterbalanced by the cost for adapting the tools for the specific context [60]. We found that large-scale MDE does not only require adapting single tools but rather requires the introduction of a MDE-specific infrastructure that compensates for the domain experts lack of software development skills. We use the term primary software to denote the software developed as part of the product to deliver – in this case a car – and the term secondary software for the software that supports the development of the primary software. Since the primary software is explorative to its nature and developed under agile forms it is not possible to define the secondary software in advance – its functionality and content will depend on the directions taken for the primary software.

We found that abstractions do not necessarily help in understanding complex software. One example is how a modelling tool enforces the user to open new windows for each level in the hierarchical structure of the software as well as for each subpart at that level. The consequence, according to one interviewee, was that you could never get the details of a part and its context at the same time which made it difficult to interpret the given model elements.

In most cases our study validates the findings of Hutchinson et al. The exceptions regard the engineers' sense of control – Hutchinson et al. found that code gurus feel they loose control while managers are risk-averse and avoid new technologies including MDE – and the lack of relevant MDE skills among practitioners. In contrast we found that the introduction of MDE in a new unit enabled the adaption of new tools, new processes and the employment of engineers that had prior experience of the chosen modelling language. And coders employed to develop secondary software saw no loss of control in their role.

While interfaces between top-level components was a problem at Volvo Cars the engineers at Ericsson had new opportunities to address the challenge due to the introduction of agile development. At Ericsson the developers get access to both ends of the interface and can change it according to the changing understanding of what needs to be communicated over the interface. The organisational impact is enabled by the software residing on one hardware unit and all software being developed in-house. As reported on in *Extending Agile Practices in Automotive MDE* [39] neither are the case at Volvo Cars so other ways of handling the challenge are needed.

In our publication regarding MDE tools [119] we claim that the MDE community needs a better way of determining when and how to introduce MDE in an organisation. From our data it seems that forming a new unit was a successful way since it allowed the engineers to adapt the experiences from other units to a new organisational context without the legacy in terms of existing culture and software.

5 Reflection

Smith defines reflection as the *"assessment of what is in relation to what might or should be and includes feedback designed to reduce the gap"* [104]. Reflection is a vital part of teaching and learning where new situations constantly arise for which we have not been specifically trained [16, 96]. But as we saw in Section 3 on research methodologies, reflection plays an important part in evaluating the decisions and outcomes of a research process. As Steeples points out, the involved researcher has the dual roles to both drive and critically review the process [110].

The four scholarships allow for different possibilities for reflection, in what Schön refers to as reflection-in-action and reflection-on-action [96]. Reflection-on-action takes place before or after the execution of an activity, at the planning or evaluation stage. Reflection-in-action is in contrast done during the activity. By changing the context the possibilities for reflection change since the time frame for reflection-in-action varies from seconds in a class room or during interviews to days or weeks during the implementation of a design. And the ability to reflect in-action which pair lecturing provided was useful during the interviews since it was necessary to both reflect on how the situation evolved while being involved in the discussion.

Different roles have different perspectives, even within the scholarships. A teacher and a student will see different challenges and possibilities and bring their own assumptions into the equation. The same goes for an engineer and a manager who will have different experiences and perspectives on software development within the same organisation. By interacting with a number of different practitioners the ability to see new possible interpretations of a phenomena grows.

In relation to the changing perspectives the level of expertise of the academics vary, both in relation to MDE and the domain it is used for. In some cases the academics' knowledge of MDE in general is greater than the practitioners, as in the class room, while in other cases practitioners have more in-depth knowledge of their way of working with MDE and the domain, while the contribution of the academic instead is to provide new interpretations. And in some case the academic becomes the novice yet again.

Transferring between the different scholarships has also enabled new perspectives on the diverse activities within the scholarships. Sometimes this has lead to publications but other times the results are more informal in terms of feedback on how to improve or handle different aspects. The collaboration behind *Executable and Translatable UML – How Difficult Can it Be?* is an example of where the discussions have led to a formal result in terms of both course improvement and publication. Another example is how the feedback from our students on executable modelling tools is the reason why the challenges facing the engineers regarding interfaces was recognised. In general, being active in more than one scholarship gives new contexts to air and discuss recurring challenges that often are shared but have different impact depending on context.

The different audiences when communicating the results of the research means insights into different research traditions such as computer science's focus on implementing designs in comparison to inducing new insights from empirical data.

6 Conclusions

The contributions under Teaching relate to how novice modellers can be helped to master the process of organising domain knowledge into software solutions. This was achieved by transitioning from models as blueprints to using models as the programming language [27]. The second change was to move from lectures focused on describing the features of the models to how we reason when using the models [23]. While the scope of the course is model-driven development the skills should be transferable to code-centric development since the focus is on achieving good designs and articulating the pros and cons of different solutions.

The main theoretical contribution for Discovery is how Natural Language Generation relates to the concept of model transformations [20]. The benefit of generating natural language texts from models instead of code is that the texts can be reused across implementations and describe the same structure and behaviour independent of how it is realised in terms of hardware, operative system and software representations. The technique is also domain-independent since the terminology of the model is reused for the textual descriptions.

Our contributions towards the Application of MDE stem from an industrial case study on multi-paradigmatic modelling within the telecoms domain. We saw how executable models allowed the independent development of distinct domains [26] while the subsequent transformations caused problems due to both the different paradigms of the source and target languages as well as from organisational perspective [54].

While raising the level of abstraction might seem a technical issue, our contributions enforce how the success of adopting Model-Driven Engineering in industry depends on both organisational and social factors as well as technical [119]. We have also shown how balancing the chosen modelling techniques with a suitable organisation empowers new user groups as software developers as well as speeds up the development process [28, 39].

Our findings also support the claims of Kent [65] and Ameller [3] that using models in software development is more than a technical question and that modelling abstraction in terms of clearcut platform-independent and platform-specific qualities is not always applicable [28, 39]. The insight manifests itself in the contributions a progression from MDA to MDE as interaction with industry puts insights from Teaching and Discovery into new contexts. An example of the transition is how the two first publications on Natural Language Generation under Discovery were positioned in an MDA context. The third publication, found under Integration, instead avoids using the terminology of MDA and instead reasons about the relationship between model and text without fixed definitions of abstraction layers.

7 Future Work

For the future we see three main areas that we would like to further pursue – industrial evaluation of natural language generation, promoting life-long learning and agile MDE.

The main concern relating to natural language generation from software models is the lack of industrial evaluation. It is not only in the case of our contributions but in the field as a whole [25]. We aim to initiate a new industrial collaboration to evaluate to which extent practitioners in industry find generated texts useful. As part of such an evaluation the generation procedure will be included to see if and how the technique could be used in general and to determine how much linguistic effort in terms of grammars is needed to fulfill the needs of the readers.

If the role of software and how it is developed keeps changing the coming twenty years as it has the previous twenty years it will be an industrial challenge to ensure that 'old' employees can keep in touch with new trends. This is already a concern that was mentioned during the interviews at the three companies where the unwillingness to adapt new technologies and paradigms was a common obstacle for systematic change. This opens for new collaborations between industry and academia for continuous learning and emphasises the impact of knowledge transfer.

In our opinion agile MDE warrants more attention. Raising the level of abstraction, automating recurring and complicated tasks as well as full-scale simulations can enable domain experts to become the primary software developers. We still don't know how this scales up in large and distributed software development and if the investment in MDE infrastructure has a positive impact on return of investment in the long run. These are questions we would like to further investigate.

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] S.W. Ambler. *The Elements of UML^{TM}- 2.0 Style*. Cambridge University Press, 2005.

[3] David Ameller. Considering Non-Functional Requirements in Model-Driven Engineering. Master's thesis, Universitat PolitÃĺcnica de Catalunya, June 2009.

[4] 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.

[5] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36 – 41, September 2003.

[6] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169 –178, July 2010.

[7] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar – An Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Richard L. Baskerville. Investigating Information Systems with Action Research. *Communications of the Association for Information Systems*, 2, 1999.

[9] John Bateman and Michael Zock. Natural Language Generation. In Ruslan Mitkov, editor, *The Oxford Handbook of Computational Linguistics*, Oxford Handbooks in Linguistics, chapter 15. Oxford University Press, 2003.

[10] P Baxter and S Jack. Qualitative Case Study Methodology: Study Design and Implementation for Novice Researchers. *The Qualitative Report*, 13(4):544–559, December 2008.

[11] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. SCRUM: A pattern language for hyperproductive software development. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 637–652. Addison Wesley, 2000.

[12] Grady Booch. *The Unified Modeling Language User Guide*. Pearson Education, 2005.

[13] Ernest L. Boyer. *Scholarship Reconsidered: Priorities of the Professoriate.* Carnegie Foundation for the Advancement of Teaching, Princeton University Press, New Jersey, USA, December 1990.

[14] Ernest L. Boyer. From Scholarship Reconsidered to Scholarship Assessed. *Quest*, 48(2):129–139, 1996.

[15] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool, USA, 2012.

[16] A. Brockbank and I. McGill. *Facilitating Reflective Learning In Higher Education.* SRHE and Open University Press Imprint. McGraw-Hill Education, 2007.

[17] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004.

[18] John Seely Brown, Allan Collins, and Paul Duguid. Situated cognition and the culture of learning. *Educational Researcher*, 18(1):32–42, Jan-Feb 1989.

[19] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[20] Håkan Burden. *Three Studies on Model Transformations - Parsing, Generation and Ease of Use.* Licentiate thesis. University of Gothenburg, Gothenburg, Sweden, June 2012.

[21] 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.

[22] Håkan Burden and Rogardt Heldal. 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.

[23] Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Model Modelling and Encourage Active Learning. In *Proceedings of ALE 2012, 11th Active Learning in Engineering Workshop*, Copenhagen, Denmark, June 2012.

[24] Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Enabling Interface Validation through Text Generation. In *In VALID 2013, 5th International Conference on Advances in System Testing and Validation Lifecycle*, Venice, Italy, November 2013.

[25] Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Opportunities for Agile Documentation Using Natural Language Generation. In *ICSEA 2013, 8th International Conference on Software Engineering Advances*, Venice, Italy, November 2013.

[26] Håkan Burden, Rogardt Heldal, and Martin Lundqvist. Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing. In *6th International Workshop on Multi-Paradigm Modeling MPM'12*, Innsbruck, Austria, October 2012. ACM.

[27] 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.

[28] Håkan Burden, Rogardt Heldal, and Jon Whittle. Comparing and Contrasting Model-Driven Engineering at Three Large Companies. Submitted to ESEM 2014, 8th International Symposium on Empirical Software Engineering and Measurement. To be held in Torino, Italy, September 2014.

[29] Nancy Cartwright. Models: The blueprints for laws. *Philospohy of Science*, 64(1):292–303, December 1997.

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

[31] L. Cohen, L. Manion, and K.R.B. Morrison. *Research Methods in Education.* Routledge, 2007.

[32] Andy Crabtree, Peter Tolmie, and Mark Rouncefield. "How Many Bloody Examples Do You Want?" Fieldwork and Generalisation. In Olav W. Bertelsen, Luigina Ciolfi, Maria Antonietta Grasso, and George Angelos Papadopoulos, editors, *ECSCW 2013: Proceedings of the 13th European Conference on Computer Supported Cooperative Work, 21-25 September 2013, Paphos, Cyprus*, pages 1–20. Springer London, 2013.

[33] John Daniels. Modeling with a sense of purpose. *IEEE Software*, 19:8–10, January 2002.

[34] Keith Davis. Methods for studying informal communication. *Journal of Communication*, 28(1):112–116, 1978.

[35] Daniel C. Dennett. Real patterns. *The Journal of Philosophy*, 88(1):27–51, January 1991.

[36] Linda Dickens and Karen Watkins. Action Research: Rethinking Lewin. *Management Learning*, 30(2):127–140, 1999.

[37] E. W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[38] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42(4):42–52, April 2009.

[39] Ulf Eliasson and Håkan Burden. Extending Agile Practices in Automotive MDE. In *XM 2013, Extreme Modeling Workshop*, Miami, USA, October 2013.

[40] Henry Etzkowitz, Andrew Webster, and Peter Healey. *Capitalizing knowledge: New intersections of industry and academia.* Suny Press, 1998.

[41] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database.* MIT Press, Cambridge, MA, 1998.

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

[43] Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, pages 219–245, 2006.

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

[45] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[46] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *International Conference on Software Engineering, ICSE 2007, Track on the Future of Software Engineering, FOSE 2007*, pages 37–54, Minneapolis, MN, USA, May 2007.

[47] Matt Germonprez and Lars Mathiassen. The Role of Conventional Research Methods in Information Systems Action Research. In Bonnie Kaplan, III Truex, DuaneP., David Wastell, A.Trevor Wood-Harper, and JaniceI. DeGross, editors, *Information Systems Research*, volume 143 of *IFIP International Federation for Information Processing*, pages 335–352. Springer US, 2004.

[48] Holger Giese, Stefan Neumann, Oliver Niggemann, and Bernhard Schätz. Model-Based Integration. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 2, pages 17–54. Springer Berlin/Heidelberg, 2011.

[49] Tony Gorschek, Per Garre, Stig Larsson, and Claes Wohlin. A model for technology transfer in practice. *IEEE Software*, 23(6):88–95, 2006.

[50] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky, editors, *WCRE*, pages 35–44. IEEE Computer Society, 2010.

[51] Richard R. Hake. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1):64–74, 1998.

[52] Cécile Hardebolle and Frédéric Boulanger. Exploring Multi-Paradigm Modeling Techniques. *Simulation*, 85(11-12):688–708, November 2009.

[53] Rogardt Heldal, Daniel Arvidsson, and Fredrik Persson. Modeling Executable Test Actors: Exploratory Study Done in Executable and Translatable UML. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference*, pages 784–789, Hong Kong, China, December 2012. IEEE.

[54] Rogardt Heldal, Håkan Burden, and Martin Lundqvist. Limits of Model Transformations for Embedded Software. In *35th Annual IEEE Software Engineering Workshop*, Heraklion, Greece, October 2012. IEEE.

[55] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[56] B. Hoffman, H. Mentis, M. Peters, D. Saab, S. Schweitzer, and J. Spielvogel. Exploring Design as a Research Activity. In *Proceedings of the 6th Conference on Designing Interactive Systems*, pages 365–366, University Park, PA, June 2006. ACM, New York, NY.

[57] Pat Hutchings and Lee S. Shulman. The Scholarship of Teaching: New Elaborations, New Developments. *Change: The Magazine of Higher Learning*, 31(5):10–15, September/October 1999.

[58] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[59] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[60] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[61] Jakob H. Iversen, Lars Mathiassen, and Peter Axel Nielsen. Managing Risk in Software Process Improvement: An Action Research Approach. *MIS Quarterly*, 28(3):395–433, September 2004.

[62] Jean-Marc Jézéquel, Benoît Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the Chasm Between MDE and the World of Compilation. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[63] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, March 2008.

[64] David Kember and Lyn Gow. Action research as a form of staff development in higher education. *Higher Education*, 23(3):297–310, 1992.

[65] Stuart Kent. Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, 2002. Springer-Verlag.

[66] 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 Transactions on Software Engineering*, 28:721–734, August 2002.

[67] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Pearson Education, 2008.

[68] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise.* Addison-Wesley Professional, Boston, MA, USA, 2005.

[69] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2012.

[70] 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.

[71] Kurt Lewin. Action research and minority problems. *Journal of social issues*, 2(4):34–46, 1946.

[72] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2:5–14, 2003.

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

[74] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20:14–18, 2003.

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

[76] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[77] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[78] Farid Meziane, Nikos Athanasakis, and Sophia Ananiadou. Generating Natural Language Specifications from UML Class Diagrams. *Requirements Engineering*, 13(1):1–18, 2008.

[79] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[80] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and MiguelA. Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, 2013.

[81] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[82] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, 2008.

[83] Michael Polanyi. *The Logic of Liberty: Reflections and Rejoinders.* University of Chicago Press, Chicago, Illinois, 1951.

[84] Michael Prince. Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, 93(3):223–231, 2004.

[85] 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.

[86] Aarne Ranta. *Implementing Programming Languages - An Introduction to Compilers and Interpreters*, volume 16 of *Texts in Computing.* College Publications, King's College, London, UK, 2012.

[87] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *27th International Conference on Software Maintenance*, pages 103–112, Williamsburg, VA, USA, September 2011. IEEE.

[88] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems.* Cambridge University Press, 2000.

[89] Colin Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers.* Regional Surveys of the World Series. Blackwell Publishers, 2002.

[90] Bernhard Rumpe. Agile modeling with the UML. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, number 2941 in Lecture Notes in Computer Science, pages 297–309. Springer Berlin Heidelberg, January 2004.

[91] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley, 2012.

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

[93] Paulo Sergio Medeiros dos Santos and Guilherme Horta Travassos. Action Research Use in Software Engineering: An Initial Survey. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 414–417, Washington, DC, 2009. IEEE Computer Society.

[94] Walt Scacchi. *Process Models in Software Engineering*, pages 993–1005. John Wiley & Sons, Inc., 2002.

[95] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[96] D.A. Schön. *The Reflective Practitioner: How Professionals Think in Action.* Number TB5126 in Harper Torchbooks. Basic Books Publ., 1983.

[97] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[98] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[99] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September 2003.

[100] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software and Systems Modeling*, pages 1–14, 2012.

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

[102] 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.

[103] Herbert Alexander Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, 1969.

[104] Ronald A Smith. Formative evaluation and the scholarship of teaching and learning. *New Directions for Teaching and Learning*, 2001(88):51–62, 2001.

[105] 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.

[106] 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.

[107] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer-Verlag, 1973.

[108] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons Inc., 2005.

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

[110] Christine Steeples. Using action-oriented or participatory research methods for research on networked learning. In *Proceedings of the Network Learning Conference*, Lancaster University, England, April 2004.

[111] L Stenhouse. What is action research? University of East Anglia, Norwich, 1979.

[112] J. Stice. *Teaching problem solving.* TA handbook. Texas University, TX, 1996.

[113] Ruth A. Streveler, B.M. Moskal, and Ronald L. Miller. The Center for Engineering Education at the Colorado School of Mines: Using Boyer's Four Types of Scholarship. In *Frontiers in Education Conference, 2001. 31st Annual*, volume 3, pages F4B:11–14, 2001.

[114] Yu Sun, Zekai Demirezen, Marjan Mernik, Jeff Gray, and Barrett Bryant. Is My DSL a Modeling or Programming Language? In *Proceedings of 2nd International Workshop on Domain-Specific Program Development (DSPD)*, Nashville, Tennessee, 2008.

[115] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary Findings from a Survey on the MD State of the Practice. In *ESEM*, pages 372–375. IEEE, 2011.

[116] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, benefits, and problems of software modelling and model driven techniques – A survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110 – 2126, 2013.

[117] E. van den Hoven, J. Frens, D. Aliakseyeu, J. Martens, K. Overbeeke, and P. Peters. Design research & tangible interaction. In *TEI'07: Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, pages 109–115, Baton Rouge, LA, February 2007. ACM.

[118] Eelco Visser. A Survey of Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[119] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem? In Ana Moreira and Bernhard Schaetz, editors, *MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

[120] Jon Whittle, Mark Rouncefield, and John Hutchinson. The state of practice in model-driven engineering. *IEEE Software*, 2013.

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