Abstract

For the development of complex software systems, two paradigms have become popular in industry: model-based development and Agile software development. In model-based development, models are the core development artifacts, particularly in early development phases such as specification and design. The short development cycles of Agile development, and in particular continuous integration, are sometimes seen as conflicting with the apparent longer development phases in model-based development. We study how software development can benefit from combining these two paradigms successfully into *continuous model-based development*.

In this licentiate thesis, we present four papers studying continuous model-based development of complex embedded systems in industry. The first two papers present investigations of the current state-of-the-art and state-of-practice of combining model-based development and continuous integration. In particular, specific challenges to the combination are identified. In the third and fourth papers, we focus on one of those challenges: model synchronization, i.e., the management of consistency between disparate development artifacts describing the same system or parts of it. We propose a lightweight approach that notifies developers of arisen inconsistency between different models. Lastly, we consider the aspect of variability among different development artifacts. In particular, we provide automated support for alleviating manual tasks in maintaining consistency across model variants organized in a product line.

Sammanfattning

Två populära paradigmer för utveckling av komplexa mjukvarusystem är modellbaserad utveckling och agil mjukvaruutveckling. I modellbaserad utveckling är modeller kärnartiklar för mjukvaruutveckling, speciellt för att uttrycka specifikation och design. De korta utvecklingscyklerna i agil utveckling, i synnerhet vid kontinuerlig integration, ses ibland som motstridiga med de längre utvecklingsfaserna i modellbaserad utveckling. Vi fokuserar på hur mjukvaruutveckling kan dra nytta av att de två paradigmerna framgångsrikt kan kombineras till kontinuerlig modellbaserad utveckling.

I denna licentiatavhandling presenterar vi fyra artiklar som studerar kontinuerlig modellbaserad utveckling av komplexa inbyggda system inom industrin. De två första artiklarna presenterar undersökningar av den aktuella situationen och specifika utmaningar för att kombinera modellbaserad utveckling och kontinuerlig integration. I den tredje och fjärde artikeln fokuserar vi på en av dessa utmaningar: modellsynkronisering, det vill säga hanteringen av konsistens mellan olika utvecklingsartefakter som beskriver samma system. Vi föreslår en metod som informerar utvecklare när inkonsistens mellan olika modeller introduceras. Slutligen undersöker vi variabilitet mellan olika utvecklingsartefakter och presenterar ett automatiskt stöd för att förenkla det manuella arbetet att upprätthålla konsistens mellan modellvarianter organiserade i en produktlinje.

Samenvatting

Voor de ontwikkeling van complexe softwaresystemen zijn twee praktijken breed omarmd: model-gebaseerde ontwikkeling en *Agile* softwareontwikkeling. In model-gebaseerde ontwikkeling staan modellen centraal, in het bijzonder in vroege ontwerpfases. De ontwikkelcycli in model-gebaseerde ontwikkeling worden vaak gezien als lang en stug, en daarmee conflicterend met de korte ontwikkelcycli in Agile methodes en nadrukkelijk *continuous integration*. We onderzoeken hoe software ontwikkeling kan profiteren van een combinatie van deze twee praktijken: *continue model-gebaseerde ontwikkeling*.

In deze licentiaat[1] thesis presenteren we vier wetenschappelijke artikelen die dit onderwerp onderzoeken in de context van ontwikkeling van complexe *embedded* softwaresystemen in bedrijven die machines ontwikkelen waarvan deze systemen deel uitmaken. De eerste twee artikelen presenteren onderzoek naar de huidige standaarden en uitdagingen, zowel in de literatuur als in de praktijk. In het derde en vierde artikel verleggen we de focus naar een van die uitdagingen: modelsynchronisatie. Daarmee wordt bedoeld, het ervoor zorgen dat verschillende artefacten (zoals modellen en programmacode), die worden ontwikkeld om een systeem te ontwerpen en te implementeren, elkaar niet tegenspreken. We stellen een “lichtgewicht” benadering voor, waarbinnen ontwikkelaars melding krijgen van ontstane tegenstrijdigheden tussen verschillende artefacten. Als laatste bijdrage in deze thesis ontwikkelen we automatische ondersteuning voor het onderhouden van consistentie tussen modellen in een productlijn, die varianten van een systeem beschrijven.

[1] Licentiaat is een Zweedse academische graad halverwege tussen een master (MSc) en een doctor (PhD).

Acknowledgment

I would like to take this opportunity to thank some of the people who were invaluable to the creation of this thesis. First of all, I would like to thank my advisors Jan Carlson, Antonio Cicchetti, and Federico Ciccozzi. Thank you for all your help during the last years; for bringing me to Sweden; for your valuable input in our research discussions; for all your patience whenever I came by your offices to talk; for the shared pearls of wisdom; for your fast and insightful feedback on drafts; and for your mentorship during conference trips (and on that note, I should also thank honorary advisor Alessio Bucaioni).

The work in this thesis was supported by Software Center. I would like to thank our contacts, interviewees, and other discussion partners in the companies we worked with, for their time and valuable input in research discussions.

I would also like to thank my IDT colleagues for many great times. I enjoyed our fika's with sweets from around the world, cinema visits, game nights, runs, and the many other activities outside work.

A few special mentions are in order. Thank you Filip for being the best office mate. Thank you Jean and Leo for accompanying me on many great adventures, disappointing or otherwise. And thank you my cycling buddy Václav for showing me many kilometers around Västerås and beyond.

Finally, but most importantly, I want to thank my family for their continued support. Especially, I want to thank my parents for everything they did for me throughout the years. *Pap en mam, bedankt voor meer dan alles.*

Robbert Jongeling
Västerås, October, 2020

List of Publications

Papers included in this thesis[2]

Paper A: Robbert Jongeling, Jan Carlson, Antonio Cicchetti, Federico Ciccozzi. *Continuous integration support in modeling tools.* In the 3rd International workshop on collaborative modelling in MDE (COMMitMDE) at the 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018).

Paper B: Robbert Jongeling, Jan Carlson, Antonio Cicchetti. *Impediments to Introducing Continuous Integration for Model-Based Development in Industry.* In the 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019). [3]

Paper C: Robbert Jongeling, Federico Ciccozzi, Antonio Cicchetti, Jan Carlson. *Lightweight Consistency Checking for Agile Model-Based Development in Practice.* In the 15th European Conference on Modelling Foundations and Applications (ECMFA 2019)[4] and the Journal of Object Technology (JOT 2019).

Paper D: Robbert Jongeling, Antonio Cicchetti, Federico Ciccozzi, Jan Carlson. *Co-evolution of Simulink Models in a Model-Based Product Line.* In the 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS 2020).

[2]The included papers have been reformatted to comply with the thesis layout.

[3]This paper was chosen among the five best papers of the SEAA 2019 programme.

[4]This paper was awarded the Best Paper Award at the Applications Track of ECMFA 2019.

Related publications, not included in this thesis

Paper V: Robbert Jongeling. "*Considerations About Consistency Management for Industrial Model-Based Development.*" In the 12th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE 2019).

Paper W: Robbert Jongeling. "*How to live with inconsistencies in industrial model-based development practice.*" In the Doctoral Symposium (DS) at the 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019).

Paper X: Robbert Jongeling, Johan Fredriksson, Federico Ciccozzi, Antonio Cicchetti, Jan Carlson. "*Towards Consistency Checking Between a System Model and its Implementation.*" In the 1st International Conference on Systems Modelling and Management (ICSMM 2020).

Paper Y: Muhammad Abbas, Robbert Jongeling, Claes Lindskog, Eduard Paul Enoiu, Mehrdad Saadatmand, Daniel Sundmark. "*Product Line Adoption in Industry: An Experience Report from the Railway Domain.*" In the 24th International Systems and Software Product Line Conference (SPLC 2020).

Paper Z: Robbert Jongeling, Antonio Cicchetti, Federico Ciccozzi, Jan Carlson. "*Towards boosting the OpenMBEE platform with model-code consistency*" In the 1st International Workshop on Open Model Based Engineering Environment (OpenMBEE) at the 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS 2020).

Contents

I Thesis **1**

1 Introduction **3**

2 Research Overview **7**
2.1 Research Goals 7
2.2 Research Methodology 10

3 Background and Related Work **13**
3.1 Model-Based Development 13
3.1.1 Model-based systems engineering 14
3.1.2 Model-based product lines 15
3.1.3 Adoption of MBD in industry 16
3.2 Agile software development 16
3.2.1 Continuous Integration 16
3.2.2 Agile model-driven development 17
3.3 Consistency Management 18
3.3.1 Relevance of consistency checking 18
3.3.2 Consistency checking approaches 19

4 Research Results **23**
4.1 Thesis Contributions 23
4.1.1 C1: Identified challenges towards combining MBD and CI 23
4.1.2 C2: Lightweight Approach to Consistency Management 24
4.1.3 C3: Change Propagation in a Model-Based Product Line 25

4.2 Paper Contributions 26
4.2.1 Personal Contributions 26
4.2.2 Included Papers 26

5 Conclusion **31**
5.1 Future Work 31
5.2 Summary 33

Bibliography **35**

II Included Papers **43**

6 Paper A:
Continuous integration support in modeling tools **45**
6.1 Introduction 49
6.1.1 Continuous Integration 49
6.1.2 Model-Based Development 50
6.2 Related work 50
6.3 Identifying relevant aspects 52
6.3.1 Core CI Aspects 52
6.3.2 MBD Aspects Related to the Core Continuous Integration CI Aspects 53
6.4 Supported aspects 57
6.4.1 Modeling Tools 58
6.4.2 Other Tools 59
6.4.3 Tool Evaluations 60
6.5 Discussion 69
6.6 Threats to validity 70
6.7 Conclusions and future work 71
6.8 Acknowledgements 71
Bibliography 73

7 **Paper B: Impediments to Introducing Continuous Integration for Model-Based Development in Industry** **79**
7.1 Introduction 75
7.2 Research approach 76
7.2.1 Context 76
7.2.2 Interview Design 77
7.2.3 Threats to validity 77
7.3 Findings 79
7.3.1 State of practice 79
7.3.2 Conservative views 80
7.3.3 Impediments 83
7.3.4 Future visions 89
7.4 Discussion 90
7.5 Related work 91
7.6 Conclusion 93
Bibliography 94

8 **Paper C: Lightweight Consistency Checking for Agile Model-Based Development in Practice** **99**
8.1 Introduction 97
8.2 Scope 99
8.2.1 Industrial context of consistency checking 99
8.2.2 Requirements 101
8.3 Our consistency-checking approach 103
8.3.1 Language consistency mapping 105
8.3.2 Model consistency mapping 109
8.3.3 Continuous integration pipeline 110
8.4 Proof of concept 111
8.4.1 Language consistency mapping 112
8.4.2 A consistency checking tool 114
8.5 Discussion 117
8.6 Related work 119

8.7 Conclusions and future work 121
Bibliography 123

9 Paper D:
Co-evolution of Simulink Models in a Model-Based Product Line 127
9.1 Introduction 123
9.2 Motivation 125
9.2.1 A clone-and-own Software Product Line 125
9.2.2 Making Software Changes 127
9.2.3 Co-evolution in the Product Line 129
9.3 Approach 131
9.4 Feasibility study 136
9.4.1 Implementation 136
9.4.2 Demonstration 139
9.4.3 Experiences 146
9.5 Discussion 147
9.6 Related work 149
9.7 Conclusion 152
Bibliography 153

I

Thesis

Chapter 1

Introduction

In the development of modern embedded systems, most innovation comes from software, leading to expressions like "this car runs on code" [1]. Hence, there has been a lot of work aiming at improving the productivity of software development and the quality of the resulting artifacts. We discuss two of the most prominent paradigms that have been widely adopted to achieve those gains: Model-Based Development (MBD) and Agile software development [2].

In MBD, models are used for the design of systems, and possibly for their implementation too [3]. Within system design, it is beneficial to abstract some of the implementation details away in favor of a more human-oriented view of structure and functionality. Models can be used at all stages of the development and for different purposes, from communication to the automatic generation of code. In this work, we use the term MBD to refer to practices in which models are used as core software development artifacts, meaning that the models are expected to undergo frequent changes and the resulting implementation is expected to be consistent with these models. We exclude from our scope those MBD practices in which models are created for temporary documentation or communication between stakeholders only. Note that this scope is thus much wider than the UML-specific focus of the Agile Modeling (AM) paradigm described by Ambler [4].

Since the publication of the Agile manifesto [2], software development has increasingly focused on shortening development cycles. Ideally, customers

are regularly presented with an enhanced implementation, allowing them to adjust the requirements and thereby the product to their needs. Important among the Agile practices is Continuous Integration (CI) [5], in which multiple developers collaborate on a software project and each of them integrates her work frequently into a shared repository. In the CI paradigm, an integration is followed by an automated build as well as automated execution of a test suite, giving the developers an up-to-date overview of the status of a project throughout the development. This allows early detection of errors and prevents a difficult integration period of uncertain length at the end of the project.

Benefits to the productivity of software development in industry have been reported for both CI [6] and MBD [7], separately. Yet, the application of both practices in combination in industrial development projects is sometimes met with skepticism [8]. We hypothesize that combining MBD and CI into *continuous MBD* can improve the productivity of software development. In this work, we study the state-of-the-art, and the state-of-practice in several industrial environments, to identify challenges to this combination. Thereafter, we provide approaches to alleviate tasks that currently involve a large manual effort and are thereby impeding the introduction of short development cycles.

In particular, we focus on tasks related to model synchronization, i.e., ensuring consistency across various development artifacts, which is a general challenge to MBD as well [9]. In a development setup where all artifacts are code, a build system typically notices inconsistent definitions between different portions of code. For example, the code will not compile when a class does not implement all the methods defined in an interface. In MBD, it often occurs that no formal links exist between artifacts, and consequently, these types of inconsistencies might go unnoticed for a long time. The ultimate consequence of this might be late changes to the implementation or even an incorrect implementation. When moving to Agile development, with its shorter development cycles and aim of continuously integrating the development artifacts, keeping them consistent becomes both more important and more challenging. Different artifacts sooner rely on each other, and hence, inconsistencies may be propagated faster across artifacts and can be more difficult to resolve. One way to address this challenge is by frequently checking the consistency across artifacts, through automated checking mechanisms.

Many approaches have been presented to deal with inconsistency across models. Existing approaches aim to satisfy different sets of requirements in various settings. We study the problem in two industrial settings. From the first of these, we obtain requirements for a consistency checking approach that can support continuous MBD. This has resulted in an approach that is less expressive and can not automatically resolve inconsistencies, but requires a lower effort than existing approaches for defining and maintaining consistency checks.

In the above introduction of consistency checking, we have focused on the consistency across different development artifacts that describe the same system, possibly at different levels of abstraction. An additional dimension of the problem emerges when the MBD setting also includes different variants of the to-be-developed system. Typically, variability is managed using software product lines [10], in which the development is based on structured reuse of development artifacts across different product variants. To preserve the integrity of the product line and the opportunities for reuse, those artifacts should be kept consistent with each other. We study this aspect in a second setting, a model-based software product line in which changes must be propagated between models describing various derived products. As with the previously discussed type of consistency checking, reducing the manual effort for this task is vital to allow for shorter development cycles and eventually, continuous MBD.

Thesis outline This licentiate thesis contains two parts. Part I is an overview of the thesis and is organized as follows. We first discuss our research process and introduce research goals in Chapter 2, after which background and related work to the thesis are discussed in Chapter 3. In Chapter 4, we provide an overview of the included papers and the contributions brought by each of them. In Chapter 5, we present conclusions and an outline of future work towards a doctoral thesis. Part II includes the collection of included papers.

Chapter 2

Research Overview

In this chapter, we introduce the overall research goals of the thesis and how we used both empirical and constructive research methods to achieve them.

2.1 Research Goals

The essence of MBD is to abstract from the implementation by capturing the problem space in models [3]. We consider MBD to refer to development practices in which models are created and maintained as core development artifacts. That is, we require models to explicitly play a central role during development and we require the implementation to conform to it.

Nevertheless, we include in our scope a broad range of MBD practices, since not all artifacts have to be models. We also consider those development settings in which graphical models play a smaller role, e.g. because textual code is written manually. For "models", we refer to system design, software design, or software implementation models. These models can be expressed in various modeling languages. Models may provide support for, e.g. automated analysis, simulation, or code generation. Furthermore, different models can be used to describe systems at different levels of abstraction. For example, a company may use MBD to capture the system requirements and structure in a SysML [11] model, whereas individual features are implemented in Simulink [12] models or code.

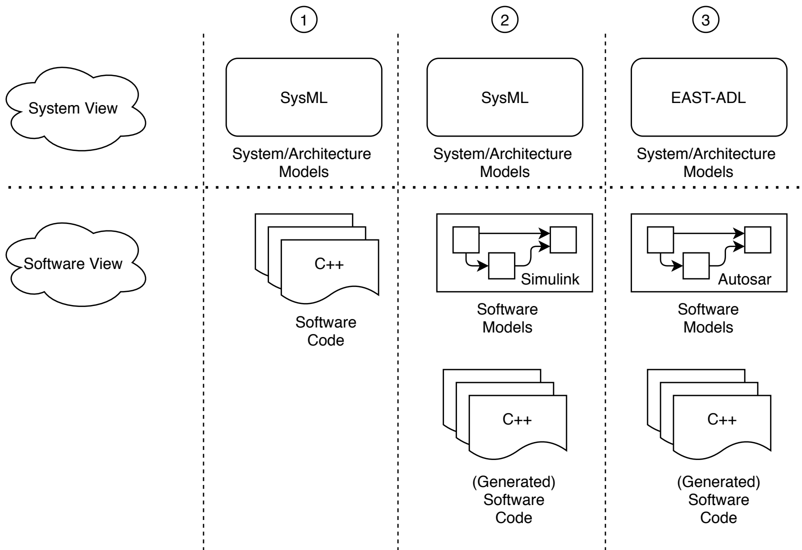


Figure 2.1. Three examples of possible artifacts in different model-based development projects.

More examples of MBD settings are illustrated in Figure 2.1. In the figure, we show three settings in which different models are used for the system view and the software view. In setting (1), the high-level system design is captured in a SysML model; this model does not contain any implementation details, but rather outlines the structure of the software as it is divided into components. Furthermore, the system model deals with concerns on dividing functionality across software components and hardware components. Hence, in this setting, code is not automatically generated from the model. In setting (2), we see that part of the implementation is automatically generated, from Simulink software models. The overall system design is again done in a SysML model and, additionally, Simulink models are created to design specific Software components. Among these example settings, there is a common need for the artifacts to be consistent with each other. Tool choices can vary across different settings too, as is illustrated by setting (3).

MBD promises to improve the productivity of software development [3].

Indeed, among the reported benefits is a reduced total development time of software systems [13]. However, MBD is traditionally viewed as "waterfall-like", with long development cycles and formal checks between each step in the cycle. After the publication of the Agile manifesto [2] and the more recent popularity gained by *DevOps*, short development cycles have become the norm in software development. One of the first steps in the DevOps paradigm and also one of the development practices promoted in Agile software development is CI. Our focus on CI is motivated by the state-of-practice at industrial partners. Due to the stringent safety requirements of their developed embedded systems, companies typically do not continuously deliver, let alone continuously deploy their software.

To illustrate continuous MBD and the CI activities on top of the existing MBD activities, consider setting ② from Figure 2.1. In the CI paradigm, the SysML model is subject to frequent changes. As a consequence, also the software models, in this example Simulink models, are subject to changes, following the updated system model. Then, also the code is generated again, following the changes to the Simulink models. As with code, also modeling artifacts should be integrated into a shared repository. The automation facilities of the CI pipeline can then be utilized to provide insight into the developed artifacts, for example through simulating the models or automatically analyzing them. The intended result of continuously integrating models too is to accelerate the feedback loop to developers, for example by allowing frequent inspection of the adherence of the code to the intended design.

To summarize, both the MBD and CI paradigms separately give improved productivity in software development; and their combination can yield additional benefits. Therefore, we propose to enhance existing MBD practices with CI features. To this end, we formulated our first research goal as follows:

> **RG$_1$:** To identify impediments towards the adoption of continuous integration in model-based development.

Towards achieving this research goal, we first identified the state-of-the-art and state-of-practice of the combination of MBD and CI. In particular, we investigated existing MBD practices in the development settings at our industrial partners and identified improvement opportunities for them. As a result, we found the need to automate some of the labor-intensive manual tasks in MBD,

so that more frequent development iterations could be established. This resulted in the second research goal, which instead aimed at identifying actions that are currently performed manually and that would need to be at least partially automated to eventually make more frequent development iterations possible and beneficial. Our second research goal was:

RG_2: To alleviate labor-intensive manual tasks that impede the adoption of short development iterations in industrial MBD settings.

2.2 Research Methodology

An old critique of software engineering research in a new guise states that "most software engineering research has the same effect on programmers that astronomy has on stars" [14]. The research community recognizes the limited practical relevance of software engineering research and suggests industry-academia collaborations as one of the means to improve it [15]. We performed our research in close collaboration with industrial partners through Software Center[1], an organization featuring 5 Swedish universities and 15 companies collaborating in software engineering research projects. The research presented in this thesis is the result of collaborations with three of the Software Center member companies, as well as two external companies.

Our research was performed in 6-month "sprints", following the commonly recommended best practice for industry-academia research collaborations of organizing the research in iterations so that research topics can frequently be fine-tuned to maximize their relevance [16]. Each of these sprints was started with a research proposal agreed upon with the partner companies. At the end of each sprint, research directions for the next sprint were proposed and results were presented in a joint workshop open to all companies in Software Center.

The iterative nature of our research process is closely related to the well-known constructive research methodology [17]. This methodology describes the common practice in software engineering research of creating knowledge by constructing solutions to well-defined problems. To identify a well-defined problem, constructive research is often preceded by empirical studies investigating

[1]https://www.software-center.se/

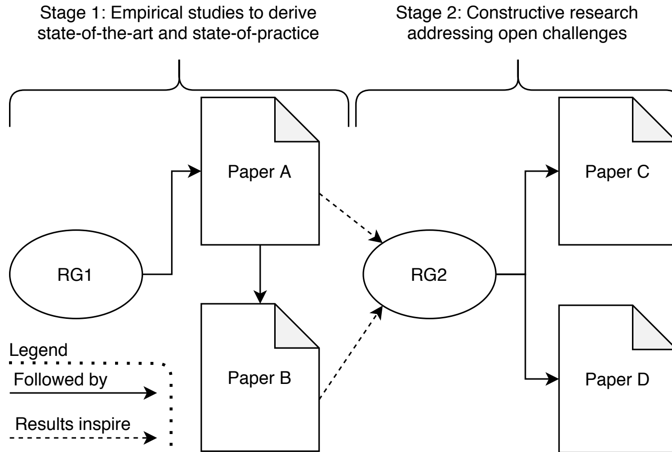


Figure 2.2. Overview of our research process

the state-of-the-art and the state-of-practice [17].

In this thesis, we present two papers (Paper A and Paper B) presenting empirical studies reporting on the state-of-the-art and state-of-practice of continuous MBD, addressing RG_1. The other two papers (Paper C and Paper D), present the construction of new approaches in collaboration with our industrial partners, addressing RG_2. Organizing the research in sprints has allowed us to use the results from Paper A and Paper B for refining RG_2 and for creating specific research projects for Paper C and Paper D. Notably, the findings of Paper A and Paper B show a lack of automated support for model synchronization, impact analysis, and co-evolution. These results have then inspired the work leading to Paper C, which targets support for consistency checking in a continuous MBD setting, and Paper D, which targets impact analysis and co-evolution in another industrial MBD setting. Figure 2.2 summarizes our research process in terms of the contributions and their interdependencies.

The aforementioned process describes the relation between our research goals and contributions. While the work addressing both research goals involved industrial partners, RG_1 aimed at identifying general research problems in the area, whereas RG_2 aimed at proposing an approach in specific industrial settings.

Therefore, in Paper A and Paper B, we used empirical methods [18] to capture the state-of-practice and identify the requirements for a new approach.

In Paper A, we describe the state-of-the-art by comparing the most-used COTS modeling tools in industry. Existing state-of-the-art literature reviews on the combination of Agile development and MBD, e.g. [19], tend to agree on high-level challenges such as tool immaturity and steep learning curve of MBD. We complement this knowledge by providing insights into available tooling, the features they support, and their shortcomings. For similar reasons, in Paper B, we conducted an interview study with practitioners. Our main objective was to identify diverse states of practice and identify open research challenges related to them. To obtain insights complementing existing literature, we involved practitioners from our partner companies. We performed semi-structured interviews [20] to provide interviewees the opportunity for personal input, while still ensuring to discuss a set of pre-defined topics.

To address RG_2, we further extended the empirical work using constructive research, in which we proposed approaches within concrete industrial settings. In this way, the collaboration differed from that in Paper B, which included interviews with engineers from several companies. Both the collaborations for Paper C and Paper D started by defining research goals for which the results of Paper A and Paper B were used as input. Upon the definition of research goals, the collaborations continued with several iterations of proposals for an approach to address the goals. Once a promising way forward had been identified, we then implemented the approach and lastly validated it.

In Paper C, we developed an approach for making developers aware of inconsistencies between models. We presented a prototype implementation and an evaluation of the approach on a limited use-case. Our evaluation indicated new requirements for a lightweight consistency checking approach. The results represent the first step towards an industry-level approach.

In Paper D, we worked in a setting in which variants of products are developed using models. We developed an approach for propagating changes from the product line to derived products. The study considered a model-based product line setting in industry, thereby making it a different setting than studied in Paper C. Hence, Paper C and Paper D are denoted in Figure 2.2 as independent papers, both originating from RG_2.

Chapter 3

Background and Related Work

This chapter contains background information and related work to the work presented in this thesis.

3.1 Model-Based Development

Several names and corresponding acronyms are in use to describe the notion of using models as key software development artifacts. Common ones include model-driven engineering (MDE), model-based development (MBD), and model-based software engineering. In our work, we refer to MBD, to emphasize that models are core development artifacts but the development includes also other artifacts such as textual documentation or code.

Figure 3.1 illustrates the four layers of the modeling stack as originating from the object management group (OMG) core specification [21]. The bottom layer (M_0, object layer) represents the real world, each of the three layers above it represents an abstraction of the layer below that layer. The first layer above the bottom, layer M_1, contains models of the real world. At this point, it should be noted that the real-world layer can also house artifacts such as code, so M_1 could contain e.g. a UML class diagram as an abstraction of some implementation. M_2, contains so-called meta-models, which denote the type of constructs that can be used to express models in M_1. UML itself is an example of a metamodel. M_3, provides meta-metamodels, the final abstraction layer since a meta-metamodel

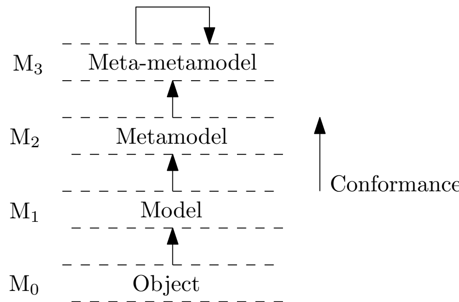


Figure 3.1. Layers of the modeling stack

describes not only instances at M_2 but also itself. Examples of meta-metamodels include Meta Object Facility (MOF)[1] and Ecore[2]. Our work is mostly concerned with instances at M_1 level and instances at M_0, but must take into account also M_2, since different instances of M_1 might conform to different instances of M_2.

Model-to-model transformations can be created to convert models conforming to one metamodel into models conforming to another metamodel. For this purpose, specialized model transformation languages have been developed, such as ATL[3] and QVTo[4]. Also, model-to-text transformations can be created, e.g. to generate code from models. Some modeling tools include such transformations and thereby support for code generation from their models.

3.1.1 Model-based systems engineering

In model-based systems engineering (MBSE), a diagrammatic *system model* is used as the central artifact containing architecture and design, thereby replacing textual documentation. The best-known language supporting this paradigm is the Systems Modeling Language (SysML), which is an extended subset of UML [11]. SysML provides a modeler with several diagrams to describe the requirements, structure, and behavior of a system [22]. Although these diagrams

[1] https://www.omg.org/mof/
[2] https://wiki.eclipse.org/Ecore
[3] https://www.eclipse.org/atl/
[4] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

may still be complemented with textual descriptions, the idea of MBSE is that the system model forms the central development artifact. This ideally enables the automatic generation of source code and documentation from the system model. In practice, automatic generation of code is not always done because it requires the model to be completed down to a very low level of abstraction, i.e., to contain a great amount of detail. Among our industrial partners, we have encountered MBSE practices in which the system model is rather a guide for the manual development of code from it. Nevertheless, also in these practices, the eventual implementation is required to be consistent with the system model, in the sense that these two descriptions of the system should not contradict each other.

3.1.2 Model-based product lines

When developing software systems, companies may need to express different versions of that system that vary on certain points. To manage this type of variability, software product line engineering (SPLE) prescribes an organization of development artifacts in product lines [10]. Various structured ways of establishing product lines are known in the literature. On the other hand, clone-and-own is an unstructured practice in which reuse is initially organized through copy-and-paste [23].

We refer to software product lines in which models are central development artifacts as model-based product lines. In general, software product lines are organized as one central development "line" from which product variants can be derived. Changes in the main product line may need to be propagated to those derived variants, for example in case of bug-fixes. The changes that need to be propagated are typically smaller than complete files. When the development artifacts are text-based, files can in most cases be merged to achieve the propagation. In model-based product lines, diagrammatic models may need to be merged, which is notoriously challenging. Moreover, the localization of the part of the model that requires propagation is not straightforward. In Paper D, we study this problem in an industrial setting.

3.1.3 Adoption of MBD in industry

Towards achieving our first research goal (RG_1), we identified challenges in combining MBD and CI. Some of the resulting challenges are shared with general challenges to the introduction of modeling practices in industry, which are reported plentiful in the literature (e.g. [24]). For example, tool interoperability, tool usability, and a steep learning curve are usual suspects among reported challenges to the industrial adoption of modeling. Furthermore, challenges are not limited to tooling issues, also human factors must be considered [25]. Despite being well-known for years, these remain open research challenges [9].

Our results offer a new perspective on these known challenges. We study those settings in which MBD has already been introduced and propose ways to make them more continuous. We expect the different perspectives to yield complementary findings and thus we also expect our results to improve the adoption of MBD in industry.

3.2 Agile software development

The manifesto for Agile software development [2] aims for customer satisfaction through frequent delivery of working software. The main effect of adopting the practices outlined in the manifesto is that development cycles become shorter, thereby allowing for frequent course adjustments. A fundamental ingredient for achieving this is Continuous Integration (CI) [5]. In the "stairway to heaven" model, CI is the third step on the evolution path from traditional engineering to continuous deployment [26].

3.2.1 Continuous Integration

In Paper A, we define CI as: "a collaborative development practice where software engineers frequently, at least daily, integrate their work into a shared repository." Besides enabling frequent deliveries of the software to clients, CI also prevents the need for a complex integration period after the implementation of all parts, which can be hard to plan for and take exceedingly long to complete. As can be seen from the definition, CI is concerned only with the development of software. The next stage is then to frequently release versions of the software

(continuous delivery). Continuous delivery in turn can be followed by frequently deploying the releases on customer devices (continuous deployment). Figure 3.2 illustrates these phases. Further extensions of the continuous development paradigm are made in DevOps (Development and Operations), in which data of the usage of the deployed software is used as input for new development iterations [27]. In this thesis, we focus only on continuous integration, not the subsequent stages.

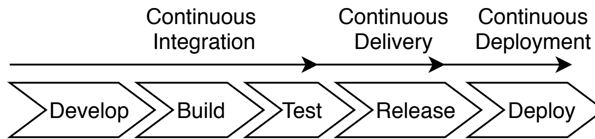


Figure 3.2. Stages of continuous development.

3.2.2 Agile model-driven development

Under the term Agile model-driven development (AMDD), several authors have presented work towards introducing Agile practices while using models as core development artifacts. Zhang et al. [28] have presented benefits of combining the two paradigms from experiences at Motorola. They present how their development processes were set up to allow for shorter development cycles, continuous integration, and frequent testing. Other case studies also find the potential benefits of applying Agile practices in MBD [29]. Rumpe has presented research results on Agile model-based software engineering using (executable) UML, presenting challenges such as model management, model composition, refactoring, and model quality [30]. Lano et al. [31] also advise a process to follow when combining Agile and modeling practices, among their tips are to do regular integration and testing. Another case study shows a successful adoption of Agile MBD and highlights the close coupling of software development with physical systems as a challenging aspect [32]. The authors address that challenge by using plant models to enable a virtual test environment, rather than relying on sparsely available physical systems, thereby contributing to shortening development increments in their model-based development. Given these experiences, there seems to be support for our hypothesis that MBD

and Agile in combination can improve development productivity. This notion is supported by the MDE research community, which has identified making modeling more Agile as one of the current research challenges [9].

We limit our focus to continuous integration (CI), one of the key practices in Agile development. Some recent work has been published towards methods and processes enabling the combination of MBD and CI. The most important of those works consider more involved modeling practices, in which models are the only development artifacts and code is generated from them. Hence, the problems they identify are closely related to that way of working. Gatcía-Díaz et al. identify model versioning and incremental artifact generation as two problems in combining modeling and CI [33]. Considering a similar level of involvement of models, Garcia and Cabot propose to utilize the continuous delivery pipeline to deal with the co-evolution of models, metamodels, and model transformations [34]. The authors propose to chain existing activities and tools using the automation capabilities of Jenkins[5].

3.3 Consistency Management

The detection and resolution of inconsistencies within or between different diagrams of the same model (intra-model consistency checking) or between different models (inter-model consistency checking) have been studied extensively. In this research, we focus on inter-model consistency checking. In particular, in Paper C, we consider consistency between different views of a system, captured in different models that are potentially created using different modeling languages and in different tools. In Paper D, we consider consistency between different models describing system variants.

3.3.1 Relevance of consistency checking

The importance of consistency in the development process is undisputed [35] but despite the considerable amount of work on model synchronization, it is still considered an obstacle to industrial adoption of MBD [36]. Industrial evaluations of multi-view modeling and its consistency problems are lacking [37], perhaps

[5]https://jenkins.io/

because of the complexity and scale of those environments. Selic identifies the scale of industrial applications as one of the main challenges to overcome for a model synchronization approach to be applicable [38]. In particular, he argues that in many cases the number of consistency links is huge, resulting in a large maintenance effort that is at constant risk of being neglected in favor of more pressing issues [38]. Another often identified challenge is a lack of tool interoperability in MBD [24], which naturally complicates the type of consistency management we are interested in. Indeed, creating traceability links between different models is required for effective tool interoperability and consistency between models [39].

Several attempts have been made to define consistency. Some of them tried to mathematically define [36] or create an ontology of possible inconsistencies [40, 41]. To arrive at a common definition, we state that views that express overlapping concerns are inconsistent when they contradict each other [42].

3.3.2 Consistency checking approaches

We now discuss several categories of existing consistency checking approaches and reflect on the existence of so many approaches while at the same time, many new approaches are still proposed.

Instant Model Synchronization

A significant amount of work has been done on approaches that promise the automatic maintenance of consistency between views. Approaches based on Triple Graph Grammars (TGGs) [43] or Single-Underlying Model (SUM) [44] establish a bidirectional transformation between different diagrams, thereby ensuring instant propagation of changes across different views of the system. Furthermore, there are many other proposed mechanisms for model synchronization, such as keeping models synchronized given a synchronized situation and traceability links [45], or automatic bidirectional synchronization derived from a one-directional model transformation [46]. Also, a hybrid approach is proposed, in which model transformations are generated for change propagation between views based on model difference, based on a common underlying meta-model for all views [47].

In our studied industrial settings, these model synchronization mechanisms are typically not applicable. The first possible obstacle is the aforementioned difference in detail captured in different models, which is particularly apparent in the described case of keeping a system model and code consistent. Furthermore, high-level system models are typically "modeling the future", i.e. the high-level models aim to describe the final product, whereas the code always represents the latest state of development. Therefore, the code is not expected to always conform to the latest version of the model. That also means that we don't want to automatically propagate (or at least not yet) changes in the high-level model. Another key reason is that we need to support the iterative and flexible development process in industry. Changes are not always definitive or fully completed. Some developers may include temporary placeholder snippets in models or code that are known to be inconsistent with other artifacts but will be resolved in later stages. In such cases, it makes no sense to try to synchronize the models, but it does make sense to make developers aware of the introduced inconsistency so that it is not forgotten about. Also, there may be artifacts we do not control, because they are third-party, open-source, or re-used from other projects. Specifically, for the SUM approach, the consequence is that some views are "read-only", i.e., they can not be changed. But furthermore, this reading may not be trivial at all, because the view could be expressed in any modeling or programming language. For these reasons, we try to formulate an approach not aimed at completely synchronizing models, but rather at an approach that allows inconsistencies but notifies the engineers when they are introduced. This follows established inconsistency tolerance ideas, which state that inconsistency must be to some extend tolerated during development such that development is not inhibited [48].

Other formalisms

Some other formalisms for detecting inconsistencies rely on common representations for different models. For example, Diskin et al. [49] propose to merge graph representations of heterogeneous models and then use the resulting single typed graph to detect inconsistencies. This approach is also an example of representing models as graphs. In another proposed approach, models are represented as graphs denoting logical facts about the models [50]. Similar to the first

approach, the graphs are then used to derive contradictions. Other approaches have been proposed in which models are represented by the operations that are needed to construct them. After this representation step, logical rules are defined to detect inconsistencies between the models [51].

Reflection

It is noteworthy that, although inconsistency challenges have been studied for many years, there are still research articles being published on the topic, which is still considered to be very challenging to achieve in industry. The result is that all proposed approaches are created with a certain set of requirements in mind that are identified as required for adoption in some particular industrial practice. Consistently, we do not pretend that our approach is universally applicable and somehow better than all the other proposed approaches of the past. Rather, we identified our own set of requirements induced by the industrial settings under study and have proposed an approach for meeting those requirements.

Chapter 4

Research Results

In this chapter, we discuss the results of our research. We first present the contributions of the thesis and how they were validated. Then, we highlight the specific contributions brought by each of the four included papers.

4.1 Thesis Contributions

This thesis presents the following three research contributions.

- C_1: Identified challenges of combining MBD and CI.
- C_2: An approach for lightweight inter-model consistency checking in continuous model-based development.
- C_3: An approach for alleviating the change propagation process in a model-based product line.

A mapping of contributions to research goals is shown in Table 4.1.

4.1.1 C1: Identified challenges towards combining MBD and CI

C_1 is brought by Paper A and Paper B in which we identified the state-of-the-art in modeling tools, the state-of-practice at several companies, and challenges

Table 4.1. Mapping of contributions to research goals.

	RG_1	RG_2
C_1	X	
C_2		X
C_3		X

towards combining MBD and CI. In the aspects of integration, building, testing, and overall automation, several relevant practices for continuous MBD were identified. We divided the identified challenges into the following categories: human, business, non-functional, and functional. Although these challenges are not specific to *continuous* MBD, some of them become more troublesome when adopting shorter development cycles. In our research, we focused on those challenges and in particular on model synchronization.

Another interesting result was discovering some MBD projects in which the adoption of CI was not seen as a good idea. This seemed to stem mostly from the existence of many manual steps in the current process, which are not easily performed at a higher frequency. Moreover, we found that those among our industrial partners that are most mature in the adoption of MBD and CI develop all their models in one single tool. This is done to avoid some of the most intimidating challenges, like tool interoperability and model synchronization.

Validation: The two papers forming this research contribution report on empirical findings on challenges in continuous MBD. To ensure the validity of these findings, several measures were taken in the design of these studies, such as ensuring a sampling of study subjects working in different roles and at different companies. More details are provided in Paper A and Paper B.

4.1.2 C2: Lightweight Approach to Consistency Management

Model synchronization, in particular managing consistency between different artifacts, arose as one of the core challenges. Hence, we studied inter-model consistency checking in industrial settings and proposed an approach for their lightweight management, within a continuous integration pipeline. This contribu-

tion is described in Paper C, which presents an approach to manage consistency between heterogeneous artifacts as well as a tool for a CI pipeline. We have contributed to the existing state-of-the-art and practice by focusing both on the continuous aspect of the consistency checks and their required "lightweightness" for usage in industrial settings. Despite their lightweight nature, our consistency checks still give useful information on structural inconsistencies.

Validation: The approach and functionality of the tool have been evaluated using an example system commonly used in the relevant literature. To evaluate the other important aspects related to usability and applicability in practice, an evaluation with industrial partners is planned as future work. In fact, we are currently working with an industrial partner on establishing these types of consistency checks between their system model and corresponding code-base.

4.1.3 C3: Change Propagation in a Model-Based Product Line

This thesis contribution is carried by Paper D. It addresses MBD settings in which multiple variants of software are developed in a *clone-and-own* product line. In such settings, changes in the product line may need to be propagated to derived products. Significant effort is spent on the analysis of the impact of changes in the product line on derived products. Our contribution is an approach for semi-automating the change propagation. We identified two benefits of our approach: (i) the analysis and change propagation process is simplified, and (ii) the approach can be used to move from a clone-and-own product line to a more structured organization of reuse across variants.

The domain expertise of developers is required to make decisions on change propagation since these choices depend on the requirements of the different products. Therefore, we instead identified the tasks with the most manual effort and provided techniques to automate those.

Validation: We evaluated the approach using publicly available models. Also, we report on qualitative results in terms of experiences of applying the proposed approach to industrial models.

4.2 Paper Contributions

Below we list abstracts and brief descriptions of the contributions of the included papers. A mapping of research contributions to included papers is shown in Table 4.2.

Table 4.2. Mapping of research contributions to included papers.

	P_A	P_B	P_C	P_D
C_1	X	X		
C_2			X	
C_3				X

4.2.1 Personal Contributions

I have been the main author and driver of the work for all included papers. The co-authors have been involved in all works through brainstorming and discussions. Furthermore, they have provided feedback on drafts of the papers.

4.2.2 Included Papers

Paper A: Continuous integration support in modeling tools.
Abstract: Continuous Integration (CI) and Model-Based Development (MBD) have both been hailed as practices that improve the productivity of software development. Their combination has the potential to boost productivity even more. The goal of our research is to identify impediments to realizing this combination in industrial collaborative modeling practices. In this paper, we examine certain specific features of modeling tools that, due to their immaturity, may represent impediments to combining MBD and CI. To this end, we identify features of modeling tools that are relevant to enabling CI practices in MBD processes and we review modeling tools with respect to their level of support for each of these features.
Paper contributions: Although the results are not surprising, the work contributes to the body of knowledge on impediments towards adopting CI in MBD.

Further, it strengthens some conclusions made previously by others that have indicated impediments such as tool interoperability and model versioning.

Paper B: Impediments to Introducing Continuous Integration for Model-Based Development in Industry
Abstract: Model-based development and continuous integration each separately are methods to improve the productivity of development of complex modern software systems. We investigate industrial adoption of these two phenomena in combination, i.e., applying continuous integration practices in model-based development projects. Through semi-structured interviews, eleven engineers at three companies with different modeling practices share their views on perceived and experienced impediments to this adoption. We find some cases in which this introduction is undesired and expected to not be beneficial. For other cases, we find and categorize several impediments and discuss how they are dealt with in industrial practice. Model synchronization and tool interoperability are found the most challenging to overcome and the ways in which they are circumvented in practice are detrimental for introducing continuous integration.
Paper contributions: The main contribution of this work is the finding that, in some of the studied settings, current practices actively inhibit companies from developing in shorter development cycles. We identify those practices and discuss how they are impeding the adoption of CI.

Paper C: Lightweight Consistency Checking for Agile Model-Based Development in Practice.
Abstract: In model-based development projects, models at different abstraction levels capture different aspects of a software system, e.g., specification or design. Inconsistencies between these models can cause inefficient and incorrect development. A tool-based framework to assist developers creating and maintaining models conforming to different languages (i.e. *heterogeneous models*) and consistency between them is not only important but also much needed in practice. In this work, we focus on assisting developers bringing about multi-view consistency in the context of agile model-based development, through frequent, lightweight consistency checks across views and between heterogeneous models. The checks are lightweight in the sense that they are easy to create, edit, use and

maintain, and since they find inconsistencies but do not attempt to automatically resolve them. With respect to ease of use, we explicitly separate the two main concerns in defining consistency checks, being (i) which modeling elements across heterogeneous models should be consistent with each other and (ii) what constitutes consistency between them. We assess the feasibility and illustrate the potential usefulness of our consistency checking approach, from an industrial agile model-based development point-of-view, through a proof-of-concept implementation on a sample project leveraging models expressed in SysML and Simulink. A continuous integration pipeline hosts the initial definition and subsequent execution of consistency checks, it is also the place where the user can view results of consistency checks and reconfigure them.
Paper contributions: Many approaches for checking inter-model consistency exist. The contribution of this work is an approach for checking inter-model consistency that is explicitly lightweight, i.e., easy to use and deploy in industrial settings. Furthermore, the approach is generic, it can be applied to any modeling language with a hierarchical structure that can be mapped onto a tree structure. It represents a first step towards creating a lightweight consistency checking approach that supports more types of structural consistency and can deal with the evolution of the involved models.

Paper D: Co-evolution of Simulink Models in a Model-Based Product Line.
Abstract: Co-evolution of metamodels and conforming models is a known challenge in model-driven engineering. A variation of co-evolution occurs in model-based software product line engineering, where it is needed to efficiently co-evolve various products together with the single common platform from which they are derived. In this paper, we aim to alleviate manual efforts during this co-evolution process in an industrial setting where Simulink models are partially reused across various products. We propose and implement an approach providing support for the co-evolution of reusable model fragments. A demonstration on a realistic example model shows that our approach yields a correct co-evolution result and is feasible in practice, although practical application challenges remain. Furthermore, we discuss insights from applying the approach within the studied industrial setting.
Paper contributions: To handle variability across different versions of devel-

oped software, product lines are adopted in industrial practice, often through clone-and-own reuse. This causes a lack of traceability and systematic re-use between variants. In this paper, we aid the hitherto manually performed process of propagating changes made in the product line to derived products. We do not anticipate the process to become completely automated in the future, but we expect this to be a step towards providing more automated means of analysis to help domain experts in their design decisions.

Chapter 5

Conclusion

In industrial practice, a broad range of MBD settings can be encountered. Across these settings, models are used at different levels of abstraction and in combination with various other types of artifacts. By advancing continuous MBD, we aim to allow companies to adopt shorter development iterations and faster feedback on their model-based designs.

This thesis presents three contributions to this goal. We first identified the challenges of combining MBD and CI. Then, we presented an approach for lightweight inter-model consistency checking in continuous model-based development. Finally, we presented an approach for alleviating the change propagation process in a model-based product line.

Both proposed approaches address model synchronization, a key challenge in establishing continuous MBD. However, holistic support for continuous MBD requires more improvements to functional aspects such as model synchronization and tool interoperability, as well as to process aspects such as collaborating on models and insight into quality metrics.

5.1 Future Work

In future research, we aim at extending and supplementing the presented approaches, and at addressing more of the identified challenges in various MBD settings, thereby further advancing continuous MBD in industry.

Extensions of approaches in this thesis. We are currently working on an industrial evaluation of the approach presented in Paper C. This might also lead to further refinements of the approach to ensure its lightweight nature. One of the starting points of Paper C is that the manual definition of traceability links across artifacts requires too much manual effort. We plan to further improve on our proposed approach in Paper C by establishing automated traceability link discovery methods. In early work towards this goal, we are working on an approach that uses information on the known structure of the model and naming conventions in model and code. Using these inputs, we aim to provide accurate suggestions for traceability links across artifacts.

The work of Paper D is planned to be extended with automated support for suggesting changes to test cases upon a change to a model. In the current setting, the majority of the test development effort is spent on ensuring test cases are still up to date after the model they cover is updated. To alleviate that effort, support for assessing the impact of model changes on test cases is needed in the first place. In the second place, for specific kinds of changes, these assessments may be improved by suggesting changes to the test cases that would synchronize them with the model again. We aim to combine this with our current work to establish a faster development cycle for models, their variants, and their test cases. This is an important step towards establishing continuous MBD in the model-based product line setting.

Support the functional aspects of continuous MBD. Additionally, we plan to address challenges identified in Paper A and Paper B that were left unsolved. In general, we work towards the goal of supporting continuous MBD in various settings. To do so, our current results need to be extended in the area of model synchronization and model management tasks. The latter includes e.g. automated analysis for change impact analysis, and model differencing and merging for parallel development by multiple, possibly geographically distributed, developers. Approaches to those challenges should furthermore be compatible with the continuous integration paradigm.

In Paper D, we incorporated the aspect of developing software variants. Variability is typically organized in software product lines, which can be used in MBD too. Models and parts of models can be re-used across different

product variants. An emerging challenge is then to manage both the dimensions of variants and revisions of these models. This is particularly pressing in component-based systems, where variation occurs on three levels since 1) the components themselves exist in different variants (or alternatives), 2) across systems, various configurations of multiple components are in use, and 3) there are various systems expressed using these components and configurations. On top of that, these variants all exist in different revisions through time. A first challenge is then to optimize the way of modeling such systems to avoid duplication of volatile information. Furthermore, consistency checking methods are needed that are aware of these dimensions and can appropriately check consistency between the appropriate versions of different artifacts.

Supporting the continuous MBD process. Supporting continuous MBD would require, in addition to solutions to technical challenges, also improvements to tooling that supports the development process of teams of developers. For code-based software development, mature tools are available for supporting activities such as code reviews and issue reporting. Mature continuous MBD practices require such supporting tooling too, that furthermore is model-aware.

Another need in industrial practice is to get insight into the quality development artifacts. In continuous MBD, these metrics should be defined and measured across different artifacts, instead of being scoped to single models or code. In this context, consistency is merely one quality metric among the typically six: correctness, completeness, consistency, comprehensibility, confinement, and changeability [52]. The continuous integration pipeline could be a good host for the calculation and presentation of quality metrics, as is typically done in dashboards for code-based software development projects.

5.2 Summary

In this thesis, we have presented contributions that are aimed at advancing continuous MBD. We have shown different setups of continuous MBD, one in which the continuous integration pipeline is utilized to include inter-model consistency checking. In Paper A and Paper B, we identified challenges to continuous MBD and in Paper C and Paper D, we presented approaches to alleviating two of these

challenges. The challenges have been identified through an investigation of state-of-the-art modeling tools and their support for continuous integration, as well as interviews with industrial MBD practitioners.

The proposed approaches are a lightweight consistency checking method and an approach to assist in change propagation within a product line. The two approaches have been defined in diverse MBD settings and in collaboration with industrial partners from different domains. Both approaches reduce the needed manual effort, which otherwise inhibits the adoption of short development cycles and, ultimately, continuous MBD.

Bibliography

[1] Robert N Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[2] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for Agile Software Development, 2001.

[3] Douglas C Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[4] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002.

[5] Martin Fowler and Matthew Foemmel. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[6] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering (Innsbruck, Austria, 2013)*, pages 736–743, 2013.

[7] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. 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, Feb 2013.

[8] Robbert Jongeling, Jan Carlson, and Antonio Cicchetti. Impediments to introducing continuous integration for model-based development in industry. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 434–441. IEEE, 2019.

[9] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, pages 1–9, 2020.

[10] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[11] Matthew Hause. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.

[12] MATLAB Simulink. The MathWorks, Natick, MA, USA.

[13] Manfred Broy, Sascha Kirstan, Helmut Krcmar, and Bernhard Schätz. What is the benefit of a model-based design of embedded software systems in the car industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369. IGI Global, 2012.

[14] Greg Wilson. Software engineering revisited. https://third-bit.com/2019/05/30/software-engineering-revisited.html, 2019.

[15] Vahid Garousi, Markus Borg, and Markku Oivo. Practical relevance of software engineering research: synthesizing the community's voice. *Empirical Software Engineering*, pages 1–68, 2020.

[16] Vahid Garousi, Kai Petersen, and Baris Özkan. Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, 79, 07 2016.

[17] Gordana Dodig Crnkovic. Constructive research and info-computational knowledge generation. In *Model-Based Reasoning in Science and Technology*, pages 359–380. Springer, 2010.

[18] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering*, pages 7–23. Springer, 2003.

[19] Hessa Alfraihi and Kevin Lano. The integration of agile development and model driven development - a systematic literature review. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, pages 451–458. SCITEPRESS, 2017.

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

[21] Object Management Group. Meta-object facility 2.0 core specification. https://www.omg.org/spec/MOF/2.0/PDF, 2006.

[22] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[23] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110, 2013.

[24] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[25] 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)*, pages 471–480. IEEE, 2011.

[26] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the 'Stairway to Heaven'–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of

Software. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 392–399. IEEE, 2012.

[27] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[28] Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE software*, 28(2):84–91, 2011.

[29] Sylvia Ilieva, Iva Krasteva, Gorka Benguria, and Brian Elvesæter. Enhance your model-driven modernization process with agile practices. In *Proceedings of the 1st International Workshop in Software Evolution and Modernization–SEM 2013*, pages 95–102. Angers Loire Valley, France, 2013.

[30] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer, 2017.

[31] Kevin Lano, Hessa Alfraihi, S Yassipour-Tehrani, and Howard Haughton. Improving the application of agile model-based development: Experiences from case studies. In *The Tenth International Conference on Software Engineering Advances*, pages 213–219, 2015.

[32] Ulf Eliasson, Rogardt Heldal, Jonn Lantz, and Christian Berger. Agile model-driven engineering in mechatronic systems-an industrial case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 433–449. Springer, 2014.

[33] Vicente García-Díaz, Jordán Pascual Espada, Edward Rolando Núnez-Valdéz, G Pelayo, B Cristina Bustelo, and Juan Manuel Cueva Lovelle. Combining the continuous integration practice and the model-driven engineering approach. *Computing and Informatics*, 35(2):299–337, 2016.

[34] Jokin Garcia and Jordi Cabot. Stepwise adoption of continuous delivery in model-driven engineering. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 19–32. Springer, 2018.

[35] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2011(E) Systems and software engineering – Architecture description. Technical report, Dec 2011.

[36] Jan Reineke, Christos Stergiou, and Stavros Tripakis. Basic problems in multi-view modeling. *Software & Systems Modeling*, pages 1–35, 2017.

[37] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software & Systems Modeling*, pages 1–27, 2019.

[38] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[39] Francis Bordeleau, Benoit Combemale, Romina Eramo, Mark van Den Brand, and Manuel Wimmer. Tool-support of socio-technical coordination in the context of heterogeneous modeling: A research statement and associated roadmap. 2018.

[40] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 356–364. IEEE, 2008.

[41] Sebastian Herzig, Ahsan Qamar, Axel Reichwein, and Christiaan JJ Paredis. A conceptual framework for consistency management in model-based systems engineering. In *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, IDETC/CIE 2011; Washington, DC, United States, 28-31 August, 2011*, pages 1329–1339. ASME, 2011.

[42] Richard Paige, Phillip Brooke, and Jonathan Ostroff. Metamodel-Based Model Conformance and Multi-view Consistency Checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):11, 2007.

[43] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[44] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Orthographic software modeling: A practical approach to view-based development. In *Evaluation of Novel Approaches to Software Engineering*, pages 206–219. Springer Berlin Heidelberg, 2010.

[45] Igor Ivkovic and Kostas Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 252–261. IEEE, 2004.

[46] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173. ACM, 2007.

[47] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. A hybrid approach for multi-view modeling. *Electronic Communications of the EASST*, 50, 2012.

[48] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[49] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying overlaps of heterogeneous models for global consistency checking. In *International Conference on Model Driven Engineering Languages and Systems*, pages 165–179. Springer, 2010.

[50] Sebastian Herzig, Ahsan Qamar, and Christiaan Paredis. An approach to Identifying Inconsistencies in Model-Based Systems Engineering. *Procedia Computer Science*, 28:354–362, 2014.

[51] Jerome Le Noir, Olivier Delande, Daniel Exertier, Marcos Aurélio Almeida da Silva, and Xavier Blanc. Operation based model representation: experiences on inconsistency detection. In *European Conference on Modelling Foundations and Applications*, pages 85–96. Springer, 2011.

[52] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and approaches to model quality in model-based software development–a review of literature. *Information and software technology*, 51(12):1646–1669, 2009.

II

Included Papers

Chapter 6

Paper A: Continuous integration support in modeling tools

Robbert Jongeling, Jan Carlson, Antonio Cicchetti, Federico Ciccozzi
In the 3rd international workshop on collaborative modelling in MDE (COM-MitMDE 2018).

Abstract

Continuous Integration (CI) and Model-Based Development (MBD) have both been hailed as practices that improve the productivity of software development. Their combination has the potential to boost productivity even more. The goal of our research is to identify impediments to realizing this combination in industrial collaborative modeling practices. In this paper, we examine certain specific features of modeling tools that, due to their immaturity, may represent impediments to combining CI and MBD. To this end, we identify features of modeling tools that are relevant to enabling CI practices in MBD processes and we review modeling tools with respect to their level of support for each of these features.

6.1 Introduction

In this work we couple two concepts: Continuous Integration (CI) and Model-Based Development (MBD). CI refers to a subset of the Agile development practices as described by Martin Fowler [1]. Several empirical evaluations have shown the positive effects of CI on productivity in industrial software development projects [2, 3]. The MBD paradigm holds the promise of increased productivity of development teams by raising the level of abstraction from code to models [4]. The benefits of MBD on productivity have been empirically assessed in industrial settings too [5, 6, 7].

We hypothesize that CI practices can further improve the productivity of MBD. In our research, we aim at identifying impediments to realizing these practices in industry. Eventually, although not in the scope of this paper, we aim at resolving them, thereby contributing to the maturity of collaborative modeling practices.

In this work, we focus on technical challenges towards introducing CI practices in MBD. We examine modeling tools to identify particular features that are commonly underdeveloped and thereby may represent potential impediments to combining CI and MBD. In particular, we answer the following research questions:

1. What are relevant features for a modeling tool to be able to support CI practices?
2. To what extent are these features provided in current modeling tools?

In the remainder of this introduction, the concepts of CI and MBD are described in more detail. The rest of the paper is organized as follows. Section 6.2 outlines related reviews of modeling tools. The first research question is answered in Section 6.3. In Section 6.4, the second research question is answered. The results are discussed in Section 6.5. Threats to the validity of our work are outlined in Section 6.6 and the conclusions are provided in Section 6.7.

6.1.1 Continuous Integration

CI is one of the twelve Extreme Programming (XP) practices [1]. In turn, XP is one of the elements of the software development concepts published in the

Agile manifesto [8]. Since then, various practices regarding the frequency and level (e.g. the entire system, a sub-system or an own branch) of software integrations have been developed [9]. The terms Continuous Integration, Continuous Deployment and Continuous Delivery are sometimes used interchangeably [10]. An unclear definition and interchangeable use of the terms may lead to their devaluation [11]. Therefore, we refer to the definition of Continuous Integration, given by Fowler [1], as follows:

Definition 1. Continuous Integration is a collaborative development practice where software engineers frequently, at least daily, integrate their work into a shared repository. After each integration, an automatic build is performed. On successful build, automated tests are performed.

In MBD, this integrated work consists of models and other modeling artefacts in addition to code. This may pose additional challenges, such as differencing on model level, that are not encountered when applying CI practices in conventional software development projects.

6.1.2 Model-Based Development

We use the term Model-Based Development (MBD) to denote modeling practices in which models are used to capture functionality and possibly to generate code. Rios et al. distinguish five maturity levels of modeling practices [12]. The levels range from ad-hoc to ultimate and describe immature to complete modeling practices. Since our goal is to introduce CI practices in MBD, it does not make sense to consider the first level, which describes immature modeling practices where models are only used by individuals for e.g. design or documentation. Instead, we are interested the more advanced levels of modeling practices where models are used by multiple team members and the eventual code or application derived from the models must be consistent with these models.

6.2 Related work

Since the publication of the agile manifesto in 2001, research on combining Agile and modeling practices [13], has been performed. Evidence-based software

engineering studies of the field have shown that Agile modeling is not a mature field yet [14, 15]. In particular, these studies identify the need for more reports on Agile modeling practices in industry.

Although Agile modeling is a topic treated in several research articles and CI is named as a needed practice in modeling [16], we have only found few that go into the details of the Agile practice of CI in combination with MBD. García-Díaz et al. identify four problems in applying CI practices in an MBD project [17]. Among these problems, the two most interesting in relation to CI (as we will see later in Section 6.3) are version control systems for models and incremental code generation. Additionally, they stress the importance of uniform, user-friendly interfaces and the variability of technologies in different phases of the pipeline. They build and evaluate a prototype solution to resolve these identified problems in an MBD project. This solution focuses on modeling approaches where models are used to generate 100% of the code for an application, whereas we consider also modeling approaches where only parts of the code are generated.

Some work has been done towards a build server for MBD [18]. The authors identify the need to support verification and validation of models. Furthermore, they argue the need for build tooling to support a mix of automatic and manual actions.

Recently, early work towards resolving impediments to combining Continuous Delivery and MBD was presented [19, 20]. The authors identify the main technical impediment to be the model-awareness of the integration server. Furthermore, they remark that for all typical MBD activities, tools that can be included in a pipeline are available. Our approach looks at this from a different perspective; we explore modeling tools and then consider the possibilities to introduce CI practices.

Naturally, there are aspects of MBD in general, not just the combination of CI and MBD, that are relevant to its adoption in industry. The impediments may be technical or be related to organizational factors of the software development process. Tooling is one of the aspects preventing a more wide-spread adoption of MBD in industry; another is the lack of clear processes to support development [21]. There are numerous organizational challenges that, if not properly tackled, hinder agility in MBD [22]. Other impediments to the adoption of MBD

that are highlighted in literature are the lack of tool interoperability [23], and the steep learning curve for developers [5, 22]. Furthermore, impediments might lie in development processes of companies and the required time and money investments to change these. Technical challenges include poor scalability of the modeling practice in general in large industrial applications of MBD [5]. Related to these are automated test generation and performance of generated code. Finally, Selic mentions challenges regarding the integration with legacy systems, traceability of generated code and the ability to execute models [24]. Nevertheless, we limit the scope of this paper to those aspects relevant for introducing CI practices in MBD.

In megamodeling, a model is created to describe the relationships between all concepts in a modeling project [25]. Since part of the work to enable CI practices is to chart the artefacts in a project and how they are related, a CI pipeline could be seen as a megamodel. In this paper, we do not further explore this relation and instead focus on existing modeling tools that are used in current practice.

6.3 Identifying relevant aspects

In this section, we identify aspects of modeling tools that are relevant for enabling CI practices in MBD. We first identify core aspects of CI based on our definition and existing literature. Then, we list particular aspects that realize these general CI aspects in the MBD domain, based on existing MBD literature. We submitted the identified aspects for review to two industrial practitioners of MBD. The resulting aspects are summarized in Table 6.1.

6.3.1 Core CI Aspects

In their literature study, Shahin et al. provide an overview of types of tools used to form a pipeline for Continuous Deployment (CD) [10]. The categories are: *Version Control System*, *Code Management and Analysis*, *Build System*, *CI Server*, *Testing*, *Configuration and Provisioning*, and *CD Server*. They note that an implementation of CD does not necessarily include all categories. Furthermore, the set of CI practices can be considered a part of that CD pipeline,

where the latter two tool categories (i.e. *Configuration and Provisioning* and *CD Server*) are excluded. Nevertheless, the CD pipeline is a good starting point to find out relevant features for combining CI and MBD.

We now elaborate on each of the proposed categories and their relevance for CI practices in MBD. A Version Control System (VCS) is used to manage different versions of developed artefacts. The artefacts are typically stored in a shared repository, sometimes allowing developers to copy it and work on their own instance, or branch. Integration of code is done into this shared repository. Code management and analysis techniques such as static code (or model) analysis might be employed to improve the quality of artefacts in a CI process. They are however themselves not closely related to the three main activities in the definition of CI: integrating, building and testing. Therefore, we do not include the Code Management and Analysis category. A build system typically combines artefacts to create executables. In the case of MBD, this could mean generating code from models but also keeping the consistency of several related models. CI servers do not perform builds themselves but rather execute builds and automated tests in other tools, the results are combined in a status overview [1]. These automated tests are important to indicate the quality of integrations. In that way, they contribute to increasing the predictability of the amount of work left, one of the purposes of CI.

6.3.2 MBD Aspects Related to the Core Continuous Integration CI Aspects

We differentiate between three types of projects to which CI can be applied. The first type is traditional software development, no models are used at all. The second type is very mature MBD, where all code is generated from models and no manual coding is performed. The third type concerns less mature kinds of MBD, where code is partially generated and then manually extended to form a complete application. We consider the latter two types in this paper. The inclusion of modeling artefacts in the CI process requires that some parts of integration, testing and building are handled differently. In this section, we identify the specific aspects of MBD that constitute these differences.

Integration

We consider the integration of changes to individual artefacts into the repository and focus on the aspects of model differencing and model merging to facilitate this integration. Integration is thus considered an activity localized to the directly changed artefacts. If multiple artefacts need to be changed to maintain consistency of the models, this is considered as the responsibility of the developer.

In order to integrate models in a shared repository, there must be a way to control their different versions. Zhang and Patel [26] refer to CI in MBD as "Continuous Modeling." They identify the need to merge frequently, but also note that merge tooling cannot handle many simultaneous changes. Merging of models is also identified as an important aspect to the adoption of modeling tools in general [27, 22]. Alternatively, pessimistic locking is used to avoid merge conflicts by allowing only one developer at a time to make changes to a model or part of a model [28].

Building

There is no direct MBD equivalent of a build system for conventional programming languages. A build system for models requires more steps than a build system for code, since code needs first to be generated from the models and possibly altered or completed by developers. Given the scope of our research, we adapt the previously identified aspect of a build server to include more model-specific actions. Automated code generation is a central part of continuous integration in a modeling context [26]. Furthermore, it is argued that code generators should work incrementally, i.e., that code should be generated only for parts of the model that have changed [17]. But the key elements of building in MBD are the ability to generate code and the ability to synchronize models and code. Therefore, we add only code generation and model discovery as relevant aspects.

For the different types of CI projects, these aspects can have different meanings. In projects with complete code generation, this generation is a task for the build server and is not performed locally by developers. When code is only partially generated, this can be done both locally and on the build server.

In case of complete code generation, model discovery is irrelevant, it will never be done since the code is never manually edited. Conversely, in a partial code generation scenario, model discovery can be needed both locally and remotely. In the simplest form, a developer locally makes changes to a piece of generated code and immediately updates the corresponding model too. In a more complex scenario, a developer could only change the code, integrate it and expect the build server to propagate her changes to the model.

Testing

We distinguish three components of testing in MBD: validating the model with respect to syntax, verifying the model with respect to predefined requirements and verifying models after integration (integration testing). The aforementioned continuous modeling practices specify unit and integration testing as important practices to build confidence in the product [26]. Verification and validation of models are identified as crucial features of a build server for MBD [18]. In both cases, "testing" specifically refers to testing the correctness of the created models, it is assumed that correct models yield correct code and thus correct applications. We therefore add model validation and verification, as well as integration testing, as sub-categories of the testing aspect.

Again, some distinctions can be made in testing between the different types of CI projects. In case of complete code generation, testing the models and the generated code should yield the same results. In partial code generation projects, code is the predominantly tested artefact. In both cases, validation of models with respect to syntax is an implicit part of the code generation process, which will not yield correct output for invalid models. This validation can also be a local action, but this is not required for the CI process. It does not prevent the integration of invalid or incorrect models, analogous to traditional software development in which e.g. non-compiling or incorrect code is integrated. In such cases, the builds or tests are expected to fail.

Model-Awareness

In some work about combining CI and MBD, authors have argued the need for more *model-awareness* in tasks related to CI. In case of the build tooling, it

is argued that manual actions that are typically performed in MBD should be taken into account and that testing should focus on validation and verification of models [18]. Others argue that the entire pipeline should be model-aware, such that dependencies between artefacts and between jobs in the pipeline that would be lost in textual representations can be discovered [20]. García-Díaz et al. also note lack of model-awareness, particularly in version control systems and code generation [17]. We do not add an aspect *model-awareness* to our list but when discussing the other aspects in modeling tools, we do take into account to what extent their implementation is model-aware.

The need for model-awareness may also refer to the need to synchronize several models when one of them is changed. In our case, this model synchronization is limited to the consistency of models and code, and between several models in a single project. In other cases, the term co-evolution is used to refer to similar activities that also include the synchronization of models and metamodels, or the synchronization of models and model transformations. Since the most used metamodels in the considered tools are UML and SysML, they and the model-to-text transformations (code generators) are typically not changed during a project. We therefore refer to this MBD aspect as "model synchronization."

Extensive support for activities related to model synchronization, such as automatically handling inconsistencies, is required in modeling tools [29]. Model synchronization is also related to code generation and model discovery, i.e., the automatic creation of a model from code. Since the generated code and models can become inconsistent after changes to either. Modeling tools can support this synchronization, e.g. by providing automated impact analysis for changed artefacts, but the process cannot always be automated. Therefore, manual actions could be required during model synchronization, this step is unsuitable for inclusion in automated builds. Since we envision a CI pipeline for models that is automated similarly to that for traditional software development, we consider model synchronization to be a task performed locally by a developer. Consequently, the CI server or build server is not concerned with tasks such as propagating changes to other artefacts. The identified need for support is still relevant, since the developer should be supported in her local work.

Automation

Interoperability of a modeling tool with other tools is an important aspect to consider too [26]. To assess their suitability to cooperate with CI servers, we look at possibilities to run modeling tools in batch mode or call their functionality from the command line. If this functionality exists, it can be used to create a script that automates part of the CI pipeline, including building and testing. Such automation is of crucial importance to the adoption of CI practices in industry.

Summary

The aspects identified in this section are summarized in Table 6.1. It contains primary aspects (in bold) and secondary, more specific aspects. In Section 6.4, we evaluate a set of modeling tools with respect to these primary aspects based on their support of the secondary aspects. Notably, not all CI aspects are directly mapped to a single MBD aspect. Rather, the MBD aspects are specific to their domain and target a more specific functionality than the general CI aspects. In the table, the citations refer to literature sources used to identify the relevance of the related aspects.

6.4 Supported aspects

In this section, we introduce the evaluated modeling tools and discuss for each tool how it implements support for the primary aspects in Table 6.1. We discuss the tools with respect to their support for the relevant aspects of CI in MBD as discussed in Section 6.3.

Note that the selection of modeling tool(s) depends on more than just the ability to use it in a CI process applied to an MBD project. Conversely, a CI process in MBD depends on more than just the used modeling tool(s), such as the maturity level of the modeling practices. The goal of our work is not to identify the *best* modeling tool for CI, but rather to investigate what impediments in applying CI to MBD projects exist.

Table 6.1. Identified relevant aspects of CI in MBD.

Aspects		CI	MBD
	Integration	[10]	[26, 27, 22]
	Model Differencing		
	Model Merging		
	Building	[10]	
	Code Generation		[26, 17]
	Model Discovery		[29]
	Model Synchronization		[29, 19]
	Testing	[10]	
	Model V&V		[18]
	Integration Testing		[26]
	Automation	[10]	[19]
	Building		
	Testing		

6.4.1 Modeling Tools

The selection of modeling tools was based on their use in industry and on inputs from our industrial partners. We included the four most-used[1] tools in industry as reported by practitioners [23]: MATLAB SIMULINK, SPARX SYSTEMS ENTERPRISE ARCHITECT, IBM RATIONAL RHAPSODY[2], and NATIONAL INSTRUMENTS LABVIEW. After discussions with our industrial partners, this list was supplemented with four additional tools that are most relevant to their daily work: NOMAGIC MAGIC DRAW, PTC INTEGRITY MODELER, ONEFACT BRIDGEPOINT, and ECLIPSE PAPYRUS. Most of these tools support UML and SysML, among the most used modeling languages in industry [23]. Exceptions to this are BRIDGEPOINT, which supports xtUML (an executable dialect of UML), LABVIEW, which supports their "G" graphical modeling language, and

[1] Excluding Eclipse-based tools and in-house tools, since they cannot be specified to a particular tool. They are reported as second and fourth most used respectively [23].

[2] In [23] the reported tool is Rational Modeler, but we include Rational Rhapsody, which can be seen as its successor.

Table 6.2. Evaluated tools in this review.

Tool	**Vendor**	**Supported Modeling Languages**
BridgePoint	OneFact	xtUML
Enterprise Architect	Sparx Systems	UML + SysML
Integrity Modeler	PTC	UML + SysML
LabVIEW	National Instruments	G
Magic Draw	No Magic	UML + SysML
Papyrus	Eclipse	UML + SysML
Rhapsody	IBM	UML + SysML
Simulink	MathWorks	Simulink

SIMULINK, which supports modeling in the Simulink language. Most of the tools thus support general purpose modeling languages. The most advanced modeling practice includes the creation of custom Domain-Specific Languages (DSLs). Tools used for that purpose are further away from the state of practice at our industrial partners and therefore not included in this evaluation. An overview of the considered tools is shown in Table 6.2.

6.4.2 Other Tools

In addition to the modeling tools, a CI pipeline typically also involves Version Control Systems (VCSs) and CI servers. There exist numerous open-source and commercial CI servers, such as JENKINS, TRAVIS, and TEAMCITY. Some of these allow for a completely custom defined pipeline whereas other tools provide users with a choice between predefined pipelines for some programming languages. Since we aim at using these tools in MBD processes, we are particularly interested in those CI servers that allow the definition of a custom pipeline. Therefore, we will mainly refer to JENKINS in the remainder of this section.

6.4.3 Tool Evaluations

We evaluated how the selected tools support the primary aspects depicted in Table 6.1. The evaluations are based on publicly available documentation and research papers about the tools. The results of the evaluation are summarized at the end of this section in Table 6.3.

Integration

In BRIDGEPOINT, version control is difficult to achieve [30]. Automated merging is not supported but the tool does show a visual difference between two versions of a model. This is not necessarily an impediment to introducing CI practices, but in practice it may discourage developers to integrate frequently if every integration potentially requires a large manual effort.

ENTERPRISE ARCHITECT (EA) has no integrated support for model versioning but relies on pessimistic locking of packages (parts of models). This system grants user exclusive editing rights on a package, thus preventing conflicts due to simultaneous changes. The tool does support the integration of several third-party version control systems, which can be used to store and manage the history of EA models. Additionally, LEMONTREE is a third-party project that supports optimistic locking, three-way merging and branching for EA models.

INTEGRITY MODELER contains a built-in service for configuration management. It includes a weak optimistic locking mechanism allowing multiple developers to collaborate on the same artefacts simultaneously. When multiple users are editing the same artefact, the changes of one of them are visible to each of the others in real-time. Alternatively, there is an optimistic locking mechanism available, where these changes are not visible. Then, merges can be performed automatically and their results manually edited to resolve merge conflicts.

LABVIEW includes a tool showing graphical differences between model versions. VCSs, such as GIT and SVN, can be used to keep track of different model versions. The differencing functionality of those tools can then be redirected to use the graphical difference available in LABVIEW. Merges can be performed automatically and merge conflicts can be resolved manually.

MAGIC DRAW contains a built-in server for version control, it provides a model repository and supports collaboration through branching and merging. Branching allows multiple developers to work in parallel on the same project. A plug-in is available to support merging at model level. In case branching is not used, concurrent changes directly on the mainline are prevented by means of pessimistic locking. The locks can be acquired at sub-model level, i.e., parts of a model can be locked for editing. The locks can then be released or maintained on commit.

PAPYRUS can be extended using plug-ins that are part of the Eclipse Modeling Framework (EMF). The Collaborative Modeling initiative provides such plug-ins, in terms of collaboration support for modeling in ECLIPSE, by using EMFCOMPARE for the detection and merging of changes, EGIT for distributed version control and GERRIT for reviews of models. This allows developers to create a branch for a project, make changes and merge them into the mainline while staying on the model level. The included version control system EGIT is an implementation of GIT, incorporated in ECLIPSE. EMFCOMPARE shows differences of changes between model versions from several views (graphical, textual, tabular). It can automatically merge changes, or in case of conflicts in three-way merges, allows the developer to choose the version to be integrated.

RHAPSODY includes the tool DIFFMERGE, which can show graphical differences and automatically merge models or projects containing models. In case of any merge conflicts, the developer is shown a graphical difference between the versions and can resolve the conflict by choosing one of the versions. The tool also supports integration of version control systems CLEARCASE and SVN.

SIMULINK provides version control support through an integrated SVN instance but can also be used together with GIT. This allows a project to be branched and thus models to be edited in parallel. SIMULINK contains an integrated tool for three-way model merging. The tool automatically merges models and on conflict offers a choice between the remote, base and local change.

Summary There are three main approaches for model versioning in the evaluated tools. The first, locking, does not scale to large collaborative projects. The second, leaving versioning completely to a VCS, is not feasible because

the VCS is typically not model-aware, which is required for merging at model level. Furthermore, line based differencing of the XML representations is not appropriate for models [28]. The third and most feasible versioning approach when introducing CI practices is to enable the integration of a version control tool in the modeling tool, but circumscribing model differencing and merging to the modeling tool itself. Indeed, this level of support is provided by multiple tools: SIMULINK, RHAPSODY, MAGIC DRAW, and PAPYRUS.

Building

BRIDGEPOINT provides integrated support for code generation. This functionality forms the "Translatable" part in "eXecutable Translatable UML" (xtUML). Models in xtUML can be transformed to C, SystemC or C++ using included model compilers. These compilers are open source and can be customized. It is also possible to create new compilers to translate models into different programming languages. Changes to generated code are not propagated back to models. So, there is only support for one-way development and not for the round-trip from models to code and back. When the generated code is a complete application rather than a skeleton or a detached, individual subsystem, this is not necessarily an impediment to introducing CI practices.

In ENTERPRISE ARCHITECT, skeleton code can be generated from both Class diagrams and Interface models. More detailed code can be generated from sequence-, activity-, and state machine diagrams. Several languages are supported, including C, C++, C#, and Java. Reverse engineering is also (partially) supported since some UML diagrams can be generated from code. ENTERPRISE ARCHITECT includes a development environment where generated code can be edited. This environment also supports typical functionalities of a code editor such as debugging and profiling. Code generation and reverse engineering can be combined and an option exists to keep models and code synchronized. When generated code is updated due to a change in the model, the body of methods is untouched, only their headers are changed such that previous work is not undone. Although ENTERPRISE ARCHITECT provides traceability matrices from requirements to models for requirements engineering, impact analysis for changes in models is not supported. Some work has been done on creating impact analysis techniques for any ENTERPRISE ARCHITECT

model, showing the potential of third party solutions to solve this problem [31].

INTEGRITY MODELER supports code generation from class and state-machine diagrams in several programming languages, including Ada, C++, and Java. The generated code might be complete but usually manual editing is required [32]. The tool includes functionality to keep models and code synchronized in real-time when manually altering the code. Furthermore, it supports impact analysis by letting the user define relationships between different modeling artefacts. These relationships can then be visualized to identify potential model elements that need to be synchronized.

Using additional code generators, C and Ada code can be generated from LABVIEW models. Templates on which the generations are based can be customized. Alternatively, the generated code can be customized after generation. Code generation is a one-way process, where the generated code is expected to be complete with no manual editing required. The generators are designed to produce code that can be integrated in a larger project.

MAGIC DRAW can generate code in several languages (Java, C++, C#). In most cases, code generated from models will be skeletons and thus will be edited by developers to implement complete functionalities. There is also support for reverse engineering; models can be derived from code. Forward and reverse code engineering is managed using *Code engineering sets*. These sets contain model elements for which code is generated and conversely files from which code is reversed to models. In addition, relationships between model components can be defined. These can be visualized in different ways to show the impact of changes on the remaining model artefacts in the project.

PAPYRUS supports code generation from UML models through plug-ins. There are plug-ins available for the generation of C++ and Java code from UML models, but it is also possible to create custom code generators for other languages. Reverse engineering is supported as part of the PAPYRUS SOFTWARE DESIGNER tooling, using it, class diagrams can be generated from Java classes and packages. Using the Papyrus Software Designer plug-in, models and generated code can be synchronized. Changes to the code are then propagated back to the model and changes in the model are incrementally applied to the code.

RHAPSODY can generate code in C, C++, Java and, using a specific RHAPSODY DEVELOPER version, also for Ada. This is done incrementally, i.e., only

new code is generated for modified model elements. The generated code can be modified and changes are propagated back to the model. It is also possible to specify code that should not be included in this round-trip, which could be useful for implementation-specific code that is not to be reused in other versions of a product. To see the impact of a change on the other artefacts in the models, RHAPSODY supports automated impact analysis. The user configures the analysis by defining, among other things, the types of links to follow and their depth. Given the result of an impact analysis, it is up to the developer to manually co-evolve the impacted artefacts.

The generation of C and C++ code from SIMULINK models is supported by additional tools that can be integrated in SIMULINK, such as EMBEDDED CODER and SIMULINK CODER. Generated code is a complete program, not just skeleton code. Modifying the generated code can be done at the level of the code generators, which can be configured to replace code by custom snippets. This also means that the process of code generation is one-way; there is no support for propagating manual changes in the generated code back to the model. SIMULINK also contains a facility for automated impact analysis that can predict impacted elements in anticipation of a particular change.

Summary The basic functionality of generating skeleton code from models is present in each of the considered tools. The detail of the created models dictates whether the entire application or only skeleton code can be generated. This distinction usually influences the functionality regarding synchronization of models and code too. This aspect is usually better supported in tools that just produce skeleton code than in tools that produce complete code and where thus the generated code does not require manual editing. We have seen that some tools contain functionality to assess the impact of changes at model level, but that the implementations still rely mostly on manual actions, which is not ideal in an automated build scenario. We note that this type of synchronization functionality focuses on models and code created in a single modeling tool. In projects where multiple modeling tools are used, more challenges related to the synchronization of the different models can be expected. This is mainly due to the limitation of impact analysis to assess only the impact of changes in models to models created in the same tool.

Testing

To test models, BRIDGEPOINT provides a *verifier*. This functionality forms the "eXecutable" part of xtUML. The verifier can be used to test models without the need to regenerate source code. It executes the model itself and supports placement of breakpoints and inspection of variable values during the simulations. This can be useful for manual testing, but less so in CI settings, where automated testing is preferred.

In ENTERPRISE ARCHITECT, models can be simulated and test scripts can be defined to automatically test model elements. In these scripts, unit tests for Java (JUnit) or .NET (NUnit) can be called. A skeleton for these unit tests can automatically be generated from class diagrams. By default, the tool supports the validation of models with respect to the UML syntax, but custom rules can be added. Furthermore, validation and test scripts can be used to automate testing of models, whereas the simulation functionality is mostly meant for debugging.

LABVIEW includes a framework for unit testing. Test cases can be defined in the tool itself by defining input values and expected output values for a specific unit under test. The tests can be executed in isolation or in a test suite. The tool includes a functionality to track tests and the code it covers, automatically providing the developer with code coverage information. In addition to this, models can be validated using static code analysis rules, which can be customized for particular purposes.

INTEGRITY MODELER contains a framework for automated testing. In it, test cases can be defined, triggered and their results viewed. The test cases can also be grouped in sessions, allowing their execution to be automated.

In MAGIC DRAW, models can be validated with respect to predefined constraints or custom created constraints expressed in the Object Constraint Language (OCL). If the validation logic cannot be expressed in OCL, boolean constraints can be defined in Java. Additionally, unit tests can be defined to verify models or integrations. JUnit is used to express test cases that can be executed using the build-in test framework. The framework also provides functionality for checking the created program for memory leaks.

PAPYRUS models can be validated with respect to predefined soundness constraints. Custom constraints can be defined in OCL. Validations can be performed on an entire model as well as on parts of a model. Warnings or errors

are shown in the models themselves after a validation. This validation is a static check, but UML models can also be executed, using the execution engine of the Moka module. This can be used to manually test models, but there is also support for automatic testing. The PAPYRUS TESTING FRAMEWORK supports the automatic generation of unit tests from UML diagrams. The unit tests can be automatically tested using the JUnit framework.

Before code is generated in RHAPSODY, a model-checker can be run to validate the model with respect to predefined and custom defined rules. Such custom rules have to be written in Java and can be used to check both the structural and the behavioral aspects of the model. Included in the tool is also a functionality to simulate models (or animate as is the used terminology for this tool). In addition, the tool can be integrated with other IBM RATIONAL tools for testing (TEST REALTIME) and quality assessment (QUALITY MANAGER). Furthermore, the tool includes a framework for the automatic generation of test cases.

Similar to code generation, there exist additional tools for the validation and testing of SIMULINK models. SIMULINK TEST is a tool that supports creation and execution of test cases for models. Test cases can be defined to verify the models with respect to functional constraints. It also provides an overview of failed and succeeded test cases, similar to the dashboard of CI tools.

Summary Most tools contain a unit testing mechanism. Some implement their own and some use existing frameworks such as JUnit. Most tools also include model validation functionality, a check of the well-formedness of the models with respect to the metamodel. Additionally, some tools are capable of simulating models, which is primarily useful for manual debugging. Next, we look at ways to automate builds and tests in the considered tools.

Automation

The BRIDGEPOINT editor is based on ECLIPSE and its model compilers are implemented as ECLIPSE plug-ins. It is possible to run these from the command line and thus incorporate them as a build step in a CI pipeline. Similarly, the testing functionality incorporated in the *verifier* can be included in an automated

process.

ENTERPRISE ARCHITECT offers the possibility to create analyzer scripts, these can be used to automate builds, tests, and other functionalities. The scripts can be created in the tool itself and allow for execution of the builds and tests from the command line. The scripts can also be used to specify an output file to contain a generated report on the test results. Furthermore, they can be used to execute the models and deploy the project, but that is out of the scope of our definition of CI.

Automation for CI can easily be achieved in INTEGRITY MODELER using a Jenkins plug-in.[3] The plug-in can detect changes in the built-in repository and can be configured to execute builds after such a detection. Furthermore, it can retrieve the results of automated tests, executed after the build. The availability of the Jenkins plug-in signals a higher level of maturity with respect to CI processes than seen in other tools.

For LABVIEW, command line interfaces are available as open-source.[4] These allow the builds and tests to be executed by a CI server, e.g. Jenkins. The test reports created by the tool can be stored as HTML files and as such be shown in Jenkins [33].

MAGIC DRAW supports extensibility by custom add-ins through its *Open API*. This API also allows the tool to be run in batch mode. This allows command line access to code generation and unit test execution, which makes it suitable to be used in a CI pipeline. Alternatively, MAGIC DRAW can also be integrated in other applications using its OSGi interfaces. Since this construct is Java-based, it is applicable in fewer cases than the generally applicable batch mode construct.

For PAPYRUS, code generation and model testing functionalities are packaged in ECLIPSE plug-ins. These are executable from the command line, which can be leveraged to include PAPYRUS in a CI pipeline, for example by calling these plug-ins from scripts managed by a CI server such as Jenkins. Creating a CI pipeline for conventional Eclipse projects is a common practice, so it is not expected that these particular tools would yield new problems.

RHAPSODY offers command line interfaces for code generation and the DIFFMERGE tool. Using these commands, code can be generated for specific

[3]https://wiki.jenkins.io/display/JENKINS/PTC+Integrity+Plugin Last access: June 4, 2018

[4]https://github.com/JamesMc86/LabVIEW-CLI, retrieved: June 1, 2018

components or for a project containing a number of components. This allows for these tasks to be integrated in a CI pipeline where only models are checked in to the version control system and the application is generated.

It is possible to create a CI pipeline using JENKINS to automatically execute builds and tests in SIMULINK. Furthermore, the CI tool can be configured to report on the success or failure of the automated tests. Such a process can be created using MATLAB, GIT and JENKINS [34].

Summary With some effort, each modeling tool can be included in a CI pipeline. We have seen some examples of ready-made pipelines including some of the discussed tools. As long as the different approaches to automation can still be executed from a pipeline, there should be no impediments regarding combining automation for multiple modeling tools in one pipeline.

Evaluation Summary

Table 6.3 summarizes the evaluations by scoring the different aspects in each tool as mature, standard, or immature. We define the thresholds for these scores, based on the identified aspects in Section 6.3, as follows. For integration, an immature level of support is considered an approach that does not allow versioning of models, but e.g. utilizes pessimistic locking. A standard support would be one where models can be merged. A mature level of support is considered when the tool also supports the visualization and resolution of conflicts. For building, an immature level of support would be one where no code generation is possible. Code generation is considered standard support, while mature support includes back-propagation of code changes to models. For testing, immature level of support only includes syntactical validation of models. Standard support involves also verification of models using model testing. Support for testing is considered mature when it includes unit tests for code and models. For automation, tools are considered to provide immature support if they do not feature explicit hooks to incorporate them in a CI pipeline. A standard level of support includes the possibility to run the tool in batch mode and perform some actions. We require the presence of a command-line API to grant mature level of support for automation.

Table 6.3. Aspects as supported by tools, scored by -, ○, or +, depicting immature, standard, or mature support respectively.

Aspects	BridgePoint	Enterprise Architect	Integrity Modeler	LabView	Magic Draw	Papyrus	Rhapsody	Simulink
Integration	-	-	○	+	+	+	+	+
Building	○	+	+	○	+	+	+	○
Testing	○	+	○	+	○	+	+	○
Automation	-	○	+	+	○	○	+	+

6.5 Discussion

We have summarized the answers to our research questions by listing relevant features for a modeling tool to be able to support CI practices in Table 6.1 and by summarizing the extent to which these features are present in current modeling tools in Table 6.3. Regarding integration, most of the considered tools provide support for differencing and merging at model level. They provide representations in various formats of model differences and allow developers to resolve merge conflicts at model level. The storage of models and change history is usually managed by a VCS such as SVN or GIT. For building, all tools provide some form of code generation. Nonetheless, there is a great variability in the maturity of what the environments can do after the code is generated. Some tools automatically keep code and models synchronized whereas in others code generation is a one-way operation. The most challenging aspect seems to be the synchronization of different models, for which most of the tools include some level of change impact analysis support. Another challenge is the synchronization of models when multiple modeling tools are used in the same software project and combined in the same pipeline. Testing is supported by each of the tools, but with different maturity levels. Finally, it is possible to automate at least parts of the build and testing processes for each of the

tools. For some of the modeling tools such automated approaches are available, whereas for others it would require custom configuration.

From the evaluations of the modeling tools, we did not find any theoretical obstacle in introducing CI practices in MBD. Some tools already have fairly good support for CI, and by cherry picking features from other tools, they could be even more suitable for CI practices. In fact, in Section 6.4.3 we have mentioned some applications of CI in each of the tools INTEGRITY MODELER, LABVIEW, and SIMULINK. On the other hand, we foresee challenges in model synchronization and automation in projects involving multiple modeling tools.

6.6 Threats to validity

In drafting both the list of relevant aspects and the list of considered tools there is an amount of subjectivity and possible bias. Tools and aspects may be omitted or be listed but less relevant. Incompleteness of the list of aspects was a threat to the validity of this work, since we aim to find impeding aspects. If crucial aspects were not included, we could not find the corresponding problems in the evaluated tools. To limit this risk, we gathered core relevant aspects by investigating existing literature on the topics of CI and MBD; furthermore, the lists were informally validated by two practitioners from industry with experience in MBD and CI. Similarly, the selection of the tools contains a bias towards UML and SysML. Other languages or modeling paradigms may yield different impediments.

Other threats are related to our chosen research methodology. The evaluations are based on publicly available documentation and research papers. This means that we did not experiment with the tools themselves to assess the aspects. The advantage of this approach is that we avoid defining a scenario to evaluate the tools, which may not fit all tools or may accidentally favor some of them. On the other hand, the inherent threat of this approach is that it does not highlight possible issues only visible when using the evaluated tools in practice.

6.7 Conclusions and future work

In this paper, we identified relevant aspects of modeling tools to support CI practices. We then evaluated eight modeling tools and assessed their levels of support for each of the aspects. In the evaluated tools, we have seen different maturity levels of support for the considered aspects. Overall, we found some challenges, but no insurmountable impediments to introducing CI practices in MBD.

The next step of our research consists in a set of interviews with MBD practitioners working at different MBD maturity levels and in different companies. Practitioners will be asked to share their views on introducing CI practices in MBD and their views on the impediments in their context. These interviews may uncover hidden technical aspects that did not arise in this work, or bring up other, non-technical impediments, such as those briefly mentioned in Section 6.5.

6.8 Acknowledgements

The authors would like to thank the industrial partners for their input in discussions about this work. This work is part of a project supported by Software Center.[5]

[5] www.software-center.se

Bibliography

[1] Martin Fowler and Matthew Foemmel. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[2] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering (Innsbruck, Austria, 2013)*, pages 736–743, 2013.

[3] Ade Miller. A hundred days of continuous integration. In *Agile*, pages 289–293. IEEE, 2008.

[4] Douglas C Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[5] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context—Motorola Case Study. In *LNCS 3713*, pages 476–491. Springer, 2005.

[6] 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)*, pages 471–480. IEEE, 2011.

[7] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. 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, Feb 2013.

[8] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for Agile Software Development, 2001.

[9] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Continuous integration impediments in large-scale industry projects. In *Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA)*, pages 169–178. IEEE, 2017.

[10] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[11] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. Continuous Practices and DevOps: Beyond the Buzz, What Does It All Mean? In *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*, pages 440–448. IEEE, 2017.

[12] Erkuden Rios, Teodora Bozheva, Aitor Bediaga, and Nathalie Guilloreau. Mdd maturity model: A roadmap for introducing model-driven development. In *Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications*, pages 78–89. Springer, 2006.

[13] Scott W Ambler. Agile model driven development is good enough. *IEEE Software*, 20(5):71–73, 2003.

[14] Sebastian Hansson, Yu Zhao, and Håkan Burden. How MAD are we? Empirical Evidence for Model-driven Agile Development. In *Proceedings of XM 2014, 3rd Extreme Modeling Workshop*, volume 1239, pages 2–11.

[15] Hessa Alfraihi and Kevin Lano. The integration of agile development and model driven development - a systematic literature review. In *Proceedings*

of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, pages 451–458. SCITEPRESS, 2017.

[16] Hessa Alfraihi and Kevin Lano. A process for integrating agile software development and model-driven development. In *Proceedings of MODELS 2017 Satellite Event: FlexMDE*, pages 412–417, 2017.

[17] Vicente García-Díaz, Jordán Pascual Espada, Edward Rolando Núnez-Valdéz, G Pelayo, B Cristina Bustelo, and Juan Manuel Cueva Lovelle. Combining the continuous integration practice and the model-driven engineering approach. *Computing and Informatics*, 35(2):299–337, 2016.

[18] Henrik Steudel, Regina Hebig, and Holger Giese. A build server for model-driven engineering. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 67–72. ACM, 2012.

[19] Jokin Garcia. Continuous model-driven engineering, 2018. Retrieved: 2018-05-14.

[20] Jokin Garcia and Jordi Cabot. Stepwise adoption of continuous delivery in model-driven engineering – extended abstract. *DEVOPS*, 2018.

[21] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *International Conference on Model Driven Engineering Languages and Systems*, pages 1–17. Springer, 2013.

[22] Stavros Stavru, Iva Krasteva, and Sylvia Ilieva. Challenges of model-driven modernization-an agile perspective. In *MODELSWARD*, pages 219–230, 2013.

[23] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[24] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[25] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd workshop in software model engineering, wisme*, pages 262–271, 2004.

[26] Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE software*, 28(2):84–91, 2011.

[27] Francis Bordeleau, Grischa Liebel, Alexander Raschke, Gerald Stieglbauer, and Matthias Tichy. Challenges and research directions for successfully applying MBE tools in practice. In *Proceedings of the 20th International Conference on Model Driven Engineering Languages and Systems*, MODELS, pages 338–343, 2017.

[28] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems (IJWIS)*, 5(3):271–304, 2009.

[29] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in model-driven software engineering. In *International Conference on Model Driven Engineering Languages and Systems*, pages 35–47. Springer, 2008.

[30] Nenad Ukić, Pál L Pályi, Marijan Zemljić, Domonkos Asztalos, and Ivan Markota. Evaluation of bridgepoint model-driven development tool in distributed environment. In *Workshop on Information and Communication Technologies conjoint with 19th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2011*, 2011.

[31] Melanie Langermeier, Christian Saad, and Bernhard Bauer. Adaptive approach for impact analysis in enterprise architectures. In *International Symposium on Business Modeling and Software Design*, pages 22–42. Springer, 2014.

[32] David Norfolk. Ptc integrity modeler — a standards-based tool for systems and software engineering. Technical report, 2015.

[33] Fredrik Edling. Using LabVIEW in a Continuous Integration Environment, 2013. Retrieved: 2018-06-01.

[34] Andy Campbell. The other kind of continuous integration, 2015. Retrieved: 2018-05-14.

Chapter 7

Paper B: Impediments to Introducing Continuous Integration for Model-Based Development in Industry

Robbert Jongeling, Jan Carlson, Antonio Cicchetti
In the proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019).

Abstract

Model-based development and continuous integration each separately are methods to improve the productivity of development of complex modern software systems. We investigate industrial adoption of these two phenomena in combination, i.e., applying continuous integration practices in model-based development projects. Through semi-structured interviews, eleven engineers at three companies with different modelling practices share their views on perceived and experienced impediments to this adoption. We find some cases in which this introduction is undesired and expected to not be beneficial. For other cases, we find and categorize several impediments and discuss how they are dealt with in industrial practice. Model synchronization and tool interoperability are found the most challenging to overcome and the ways in which they are circumvented in practice are detrimental for introducing continuous integration.

7.1 Introduction

For the design of complex modern software systems, model-based development (MBD) is often leveraged, i.e., models are used as core artifacts for activities like system design, simulation, and code generation [1]. The models are core artifacts in the sense that the eventual code and application match them and the models are not used, for example, just for informal communication. Industrial practice implies collaboration on these models by multiple engineers, possibly from multiple domains. Empirical results show improved productivity of development in industrial settings when that includes using models in this way [2, 3, 4, 5].

In parallel, Continuous Integration (CI) has also been evaluated in industrial settings and shows an improvement in the productivity of software development [6, 7]. CI proposes a collaboration in which developers frequently (at least daily) integrate their work into a shared repository [8]. Notably, CI does not entail continuously making a release available (as in continuous delivery) or continuously deploying the software on user machines (as in continuous deployment). The respective scopes of these practices are illustrated in Figure 7.1.

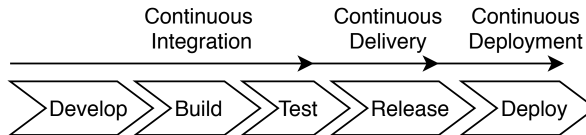


Figure 7.1. Steps included in continuous integration, continuous delivery, and continuous deployment.

To optimally make use of their individual benefits, we consider the combination of CI and MBD. This entails multiple developers using models as core artifacts for development and frequently integrating new versions of these models into a shared repository. In our experience, industrial adoption of this combination of practices is low, despite several evaluations showing its potential benefits [9, 10]. More specifically than agile, combining CI and MBD is also identified as a promising practice towards increased development productivity [11, 12]. Understandably, introducing these development methods in industry is not an overnight process and many obstacles are encountered. To identify the most important of these impediments, we have interviewed industry practition-

ers who share their experience and expectations for the future from different perspectives.

The remainder of this paper is organized as follows. In Section 7.2, we describe the design of the interview study, the findings from which are then presented in Section 7.3 and further discussed in Section 7.4. Relevant related work is considered in Section 7.5 and the paper is concluded in Section 7.6.

7.2 Research approach

7.2.1 Context

We consider the combination of CI and MBD to entail a practice in which models are developed in rapid iterations and developers integrate their work frequently into a shared repository. When models at different levels of abstraction are created, for example for system design and software implementation, they should be synchronized. Changes to any model could incur a build and test run, as part of the CI pipeline. A common current industrial practice is development using the "V-model" [13], so initially this CI can be seen as an enhancement of some steps in that process, as illustrated in Figure 7.2.

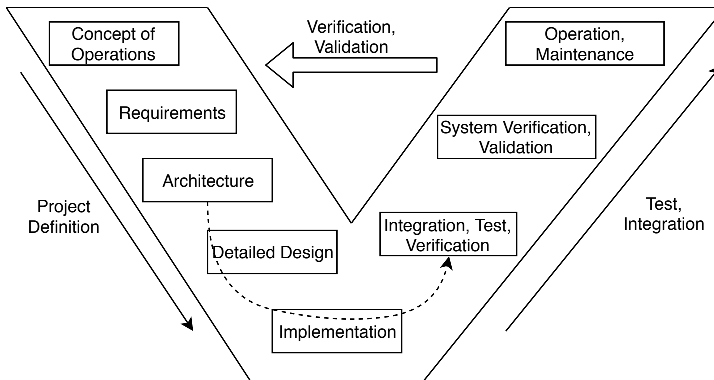


Figure 7.2. Continuous Integration concerns the steps in the bottom of the V-model [13]. Modelling artifacts belonging to these blocks may be subject to rapid iterations that may impact modelling artifacts in the other blocks, as indicated by the dashed arrow.

By means of interviews with industry practitioners of MBD, and to some extent also CI, we reflect on the difficulties of adopting this combination of CI and MBD in practice. Specifically, we aim to answer the following research question:
What are the experienced and perceived impediments to introducing continuous integration in model-based development projects?

7.2.2 Interview Design

In this work, we consider the perspective of three large companies, referred to as company 1, company 2, and company 3. Two of the companies are based in Sweden and one is based in the Netherlands. The companies develop software for embedded systems in the varied domains of avionics, electronics, and vehicular embedded systems.

We have interviewed eleven engineers, five at company 1, two at company 2, and four at company 3. The interviewees have various roles in the companies, such as system architect, system designer, software engineer, or software integrator and have varied levels of experience, from an interviewee being involved in modelling only since its company introduction in the last 2 years to an interviewee who has been involved with modelling for more than 25 years.

At two companies, the interviews were conducted by two interviewers, at the other company the interviews were conducted one-on-one. The interviews at one of the companies were in person, the others remote, through Skype and by phone. Audio of eight interviews and detailed notes of all interviews were recorded. The notes were summarized and sent to the interviewees for confirmation and discussed extensively between the interviewers.

In the design of the interviews, we have included some measures to alleviate threats to external, construct, and internal validity as well as threats to their reliability, using the categorization of these threats by Runeson and Höst [14].

7.2.3 Threats to validity

A threat to the external validity of this work is unjustly drawing generalizing conclusions based on a too narrow sample. We alleviate this threat by including companies who implement MBD to different degrees and thus have the required

different perspectives. Albeit a small sample size, we synthesize our results such that they are nevertheless relevant for companies that have adopted agile and MBD to similar degrees as the interviewed companies.

To alleviate threats to construct validity, in particular the threat of the interviewers and the interviewees having a different idea in mind when talking about agile MBD, we presented our views on these concepts before starting each interview. In this brief introduction we presented definitions of MBD and CI, as well as what we mean by their combination, similar to the introduction of this paper. Furthermore, during each interview, the interviewers have taken detailed notes that were, as mentioned, summarized and sent for review to the interviewees, such that they could confirm that we reflected their statements correctly or to allow them to correct misunderstandings.

The internal validity of our study can be impacted by interviewers not having a complete picture of the context in which the interviewee is working. Consequently, the interviewers could unjustly attribute certain impediments to certain practices. To alleviate this threat, we started the interviews with questions to determine the current state of practice at the companies and used this information after the interviews to discuss the results between the interviewers to understand from what viewpoint the different comments originate.

Towards the reliability of any interview study, there is a trade-off between a strict set of completely reproducible closed questions and an open conversation highly influenced by the personal input of the interviewers. We have balanced these interests by creating semi-structured interviews following the pyramid model [14]. Early questions are closed and specifically related to the current state of practice at companies and the professional background of the interviewees. In later stages of the interview, the questions are open and only the themes decided beforehand. During that stage, we ask the interviewees to identify impediments to combining CI and MBD, which allowed us to then talk more in-depth about these identified impediments by asking follow-up questions to explore them further.

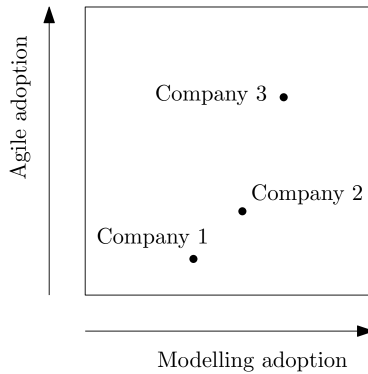


Figure 7.3. Relative positioning of involved companies based on their adoption of agile and MBD practices.

7.3 Findings

In this section, we present the results from the semi-structured interviews. First, we briefly describe the current state of practice at the industrial partners. Then, we categorize perceived and experienced impediments as identified by interviewees.

7.3.1 State of practice

All three companies are, to different extents, model-based development practitioners and also have adopted agile development practices to different extents. Figure 7.3 sketches the relative positions of these companies on both spectra, where the horizontal axis denotes adoption of models in development and the vertical axis denotes adopting agile practices with respect to the development of those models.

Company 1 utilizes models to create system designs at a high abstraction level. At this level, the design is mostly concerned with subdividing the system into clusters of software components and specifying their interfaces. After this stage, the models are handed over to software teams, who implement the designed components in code. There are some agile processes in place in the software development part, but not involving the development of the system

models. Iterations of the models are spread out over a long time and are usually made far in advance of the software development, making system models and the software implementation quite decoupled. Changes to the system design are communicated by system designers to software engineering teams after each new iteration by means of a manual handover, in which the teams agree on the changes and their implications.

Company 2 works in a similar way; a system design is created using models at a high level of abstraction, which are not directly linked to the software implementation. Additionally, a portion of the software is implemented in models from which code is generated. CI is used in the development of these software models, it is done in rapid iterations and they are integrated very frequently, up to multiple times per day.

Company 3 has a continuous integration pipeline in place for software models, all code is generated from them. For the design of the system, models are used in a less formal way, embedded in documents and only meant for communication of the design ideas. Since they are used in this way, the models do not necessarily strictly conform to their language syntax and no explicit links exist between these design models and the software models.

It should be noted that we considered only companies who already employ models for the development of their software systems. So, we are investigating only how to adopt CI once modelling is to some degree in place, rather than considering also the symmetrical case, where CI is in place but models are not used at all. Given the known challenges in adopting modelling, it is more in line with industrial practice to consider the introduction of CI in MBD, rather than supposing that modelling is introduced in an already agile, but code-centered software development process.

7.3.2 Conservative views

Early on in our research, we encountered MBD setups in which there is no need nor desire for more rapid development iterations, contradicting our initial assumptions by saying for example that there is no need for CI and that the current practices are good enough. An example of such a case involves a clear distinction between the design and implementation phases. In the design phase, system models are used to describe the system at a high level of abstraction. In

the implementation phase, models or code are used to design the software, but there are no formal or automated links between the system models and the implementation models or code. Therefore, in both of these cases, the synchronization between design and implementation is done manually, either by communication between system design teams and implementation teams, or by having the same engineers work on both parts. Rapid iterations are then undesirable because they would only increase the communication overhead between teams, or impractical, when both phases are performed by the same engineers but the implementation phase starting after the design phase is completed.

To better align our questions with these practices, we have first asked interviewees about their views on potential benefits to introducing more models in the software development, we then asked them to think about impediments they see towards that goal. Having established first the introduction of models throughout development, we moved on to questions regarding the benefits of applying CI in modelling contexts and perceived impediments towards doing so. The questions thus follow the expected adoption process of agile modelling, i.e., first introducing MBD, then going more agile. This order of the questions was chosen to separate concerns interviewees might have regarding the introduction of modelling from the impediments they encountered or expect to encounter when introducing CI.

Using more models in development can refer to extending modelling practice from system design to software models. Alternatively, it can refer to using models more formally in all development stages, rather than for example only for communication of system design concepts. The opinions are divided on the formality required in the created models, where more strictness is seen as a must for introducing more automation, but on the other hand, some engineers would like to have more freedom in a modelling tool, for example by writing free text in models, to enable easier communication of design ideas.

When code is written manually, interviewees view the potential productivity gains of introducing CI in MBD as minimal. But at the same time, modelling for code generation encounters some apprehensive views. Firstly, enabling code generation requires modelling to a low level of abstraction, which is a big step when the current practice includes only modelling on system level. This causes some reserved reactions from engineers stating that modelling is more difficult

than coding for certain concepts, such as parallelism: *"Things are sometimes very difficult to describe in models, perhaps software on a higher control level can be described, but parallel processes and what is described in VHDL on our FPGAs, . . . , complete parallelism is impossible to describe, at least in SysML. All those continuous flows, I don't know how to describe those."* Similarly, some doubts are shed to the applicability of modelling in their domain, or for their specific products: *"code generation is good if you want to make very simple things."* These types of comments illustrate some of the conservative viewpoints in industry, but more experienced modellers have expressed their wonder at the resources spent on writing code while it could be generated from models: *"I don't see why we have to write code anymore."*

More rapid iterations in the system design might also not be considered relevant because the design is often made far in advance of the software implementation and is not so flexible but only updated for very relevant changes. Another viewpoint is that it would hinder the freedom of developers, as said by one interviewee about introducing more formal modelling at system design level (instead of using models only for communication): *"I think in some situations this might help a little, but mostly it would be experienced as a limitation if those high-level diagrams have to be correct."*

CI for models at the implementation level is in place in two of the companies in this study, although those models are not explicitly linked to the system design models. It is noteworthy that both companies use, for this stage, a single modelling tool, simplifying the implementation of a CI process. More rapid iterations involving both the system design and the software implementation are not considered useful due to the previously mentioned factors. The system design is typically to a large extent completed before the implementation starts and is often created for the purpose of communication, so the models are not a precise description of the design. Furthermore, as discussed before, the process of synchronizing between the phases often involves communication between people, which scales badly to faster iterations. Rather, a more explicit coupling between the models at the different abstraction levels is desired, such that the impact of changes in the design are clearer to the system designers and the required changes in the implementation are easier communicated to the implementation teams. As reflected by the following statement about more rapid

iterations between models on all levels, *"If the models would be more connected, then that would work much better"*.

7.3.3 Impediments

The conservative views of some interviewees were not shared by all interviewees in companies 1 and 2. Some engineers did see benefits in moving towards CI even with current modelling practices. *"Yes obviously, we would like to have that, because it's working quite well in the software area, but it's hard to get to a point where it is convenient, especially in the modelling world."* In general, the benefits of a short turnaround loop between software and system design are appreciated, but thinking about doing CI in MBD may be a bridge too far. Many other problems need to be solved before getting to a stage where these practices are useful enough to be increasing productivity. It seems that the views of the engineers with conservative views to introducing CI in MBD are influenced by the impediments they foresee towards its implementation.

We have collected impediments from different perspectives, by asking engineers in companies 1 and 2 what they perceive as impediments now, and expect as impediments in the future, towards implementing CI in MBD. Engineers in company 3 were asked to reflect on their implementation of CI and the biggest obstacles encountered, as well as looking ahead to expected impediments when further streamlining their existing development processes. We categorized the obtained impediments as those having causes based on the desired functionality of the combination of CI and MBD, causes related to non-functional elements, such as the development process, human causes and business causes. A summary of those findings is provided in Table 7.1. We now elaborate each category and each identified impediment.

Functional

A CI practice in which models at multiple levels of abstraction are included requires a tight coupling between all models, such that automatic builds and tests can provide insight into the state of the integration. Since models describing the system at different levels of abstraction and from different disciplines are typically expressed in different languages and created in different tools, this

Table 7.1. Summary of identified impediments to introducing CI in MBD in this paper, in different categories of causes.

Functional	Lack of tool interoperability Lack of synchronization between models Requires model validation Lack of model merge support Lack of configuration management
Non-functional	Too long time needed for builds and tests Lack of impact analysis Tooling frustrations
Human	Lack of modelling expertise Lack of willingness to model Difference in modelling styles
Business	Difficult to enthuse management and colleagues Lack of time/knowledge to set up tool-chain Domain-induced complications

requires multiple tools that work well together. While engineers endorse this need: *"An integration would be good to have, between different tools, different categories of system designs"*, in practice, getting tools to cooperate is very challenging. *"I don't really believe strongly in having several tools if they are not really tightly integrated, but then it's the same tool."* As another interviewee stated, on the impediments to more frequent integrations of models on all levels: *"one reason is the different tool vendors, which are not integrated."* Companies have dealt with these tool interoperability challenges mostly by avoiding it. The models from which code is generated are all created in the same tool. Other models, for example system models, are created in different tools, but are connected to each other only informally, so there are no automatic checks between them or formal definitions evaluated to check that these models express the same design.

Indeed, this lack of tool interoperability consequently contributes to development in which the synchronization between models is a manual task, increasing overhead and decreasing the ability of continuously integrating new model

changes. As discussed earlier, a more explicit coupling of these models is desired. As one interviewee underscored with the following statement about system and software models: *"We have a need to integrate it really."* Or, particularly about generation of code and the creation of a feedback loop between models at different levels: *"It would be good to have a nice turnaround from design to code and from code to design. I think we will always need the possibility to change code. Because of performance reasons or maintenance reasons.* In a CI context, the state of the integration should be known after each build, but this is greatly complicated if there is no automatic support to synchronize models or check consistency between them. *"Using more models would be hard if there are no consistency checks, because if they are standalone they will never be the same."* Furthermore, mistakes due to model inconsistencies can be propagated more rapidly in a CI context.

Towards the same goals of creating an automated pipeline for building and testing, model validation is required, but typically minimally in place. This refers to syntax checking within models, since automation requires more formality from the included models, as well as consistency checking between models. The former is often in place inside modelling tools, but are not always used: *"Continuous validation of the model would be very convenient, the first step would be to get help from the tool in validating. Just get rid of all these stupid errors that you might introduce. For example, scoping, some of these errors have really serious consequences."* When models are used only for communicating designs, even less strict validation is desired by some engineers, for example the ability to write free text in some models. Consistency checking between models is hampered by the previously mentioned impediment of tool interoperability, but not felt so strongly since the models are not explicitly coupled. So, inconsistencies have no direct effects in terms of failing builds or tests, but rather may subtly affect the subsequent development, possibly incurring late and costly changes if they are noticed late. The validation of system design models is completely manual, since they are not connected to the implementation models. These manual actions do not scale to larger models or more rapid iterations and are thus hampering the move towards CI.

A common impediment to introducing MBD in industrial settings is the lack of good version control systems for models, one part of the broader challenge of

collaborating effectively on models. As one interviewee said: *"for us system engineers, this is one of the hardest parts, to share the model and not interfere with each other more than we have to."* Different to code, line-based diffs of XML representations of models are not helpful in indicating the differences between models and merging them. Particularly, the graphical representations of models are difficult to version, which is problematic especially when models grow large (and they typically do). Engineers mention a lack of model differencing in current practice: "*We have no good ways to merge models. It's even worse, we have no good ways of comparing models."* As well as too many manual steps required to obtain differences. *"There is no really good way to get a delta out of the model. To get that delta the system engineer manually has to go through and mark what is changed. To make a diff on the model, we don't have any good tools to do that."* In practice, the lack of faith in merging has two consequences. First, merging is circumvented by locking models for changes, thus avoiding the need to merge. Second, the system design is divided and tasks are assigned in such a way that the need for concurrent changes to the same model are avoided as much as possible. Locking models is a good enough solution for small teams, but already involved *"a lot of legwork"*, where intense communication was required between engineers to allow synchronous collaboration. So, locking does not scale well to larger and possibly distributed teams. It also impedes the introduction of more rapid iterations on the models. The strict division of the model is similarly impeding the agility of development. Rather, each contributor should be able to make changes at any place in the system. This lack of support for merging drives both these sub-optimal and non-scaling development practices. Notably, configuring the CI pipeline for automatic merging was also named as the biggest overcome challenge in company 3 and something that still can be improved to allow for more parallel work.

Version control is one of the components of configuration management, which manages among other things the change history and deployment of specific versions of the software on specific platforms. In general, in software engineering, configuration management is challenging, especially when software needs to be supported for a long time (possible decades) after it is first produced, thus requiring the possibilities to make changes and test them in old configurations. This is also a challenge in MBD, and a problem when trying

to do MBD in more rapid iterations, since the tooling ecosystems are typically fragile. *"I think configuration management and better tooling are what we need."*

Non-functional

In this category, we include those impediments that are not directly related to current tooling or other technical problems, but are rather related to non-functional elements such as current practices and development processes. A first example of a current practice hindering the introduction of more frequently integrating is the duration of builds and regression tests. In a similar vein, a large amount of computation power is required for extensive simulations including all models. Initially, such problems can be (and have been, by company 3) avoided by allocating more computing power to these tasks. But this is of course addressing the symptom and not the cause, and will eventually also be insufficient. This is not an insurmountable problem, but it is one of the practical hinders that are encountered when introducing agile MBD processes.

Another such practical problem is partly caused by the current division of development teams between system modellers and software designers. Changes in a system model impact lower level models, but to the system modellers, it is not always clear how. This also works the other way around; the software designers are not aware of the exact changes in the system model and the entailed required changes in software models. Both effects are strengthened by the size and complexity of the models: *"the (system) model is complex organized, the developers don't know where to look for information."* Consequently, communication, sometimes through documentation, is needed to align the activities of different modellers. This also does not scale well to larger settings and more rapid iterations. The alternative, more formal and tool supported impact analysis, requires a more formal usage of the models. If the system models are only used to communicate design and do not strictly adhere to some syntax, or do not capture the exact semantics, then trying to automatically assess impact of changes is hopeless.

Naturally, an implementation of CI will require several tools, current manual practices are not good enough to just be done more frequently. *"There are too many manual steps, too little automation."* It is therefore not a good sign that

already, often ventilated frustrations have to do with tooling. *"It is slowing us down."* Or, as indicated by another quote regarding using modelling in more phases of development: *"The modelling tool is not so stable, it crashes and it freezes and everything goes slow. You want to have quicker tools, if it was quicker and easy to understand then you could use it more."* While tool instability is not a hinder only to introducing CI, since it also hindering current MBD practices, it is still relevant to mention here, since the potential benefit of introduction of CI in MBD depends heavily on tool support. These are comments about single modelling tools and contribute to a skeptical view about involving more automation in the development process. *"People don't want to use 5 to 10 different tools."* One interviewee described current CI practices, for non-MBD projects, as involving *"a lot of small steps and something is always broken."* Given earlier comments on tool interoperability and tool instability, the interviewees seem not to expect that this is getting any easier when setting up a CI pipeline for MBD.

Humans

When discussing these functional and non-functional impediments, we cannot overlook the human aspects, which remain present even if perfect tools are created and used in the perfect process. Most of these impediments have to do with the inherent complexity of modelling, *"not enough people know SysML."* Furthermore, a steep learning curve needs to be overcome to start contributing to models, *"it is hard to learn how to model, it takes time to be a good software modeller and it is even harder to be a good system engineer."* Besides this general modelling knowledge, the complexity of the product sometimes just requires a lot of experience. *"Some parts require so much domain knowledge that you probably need to work here for ten years before you can contribute."* This contributes to a lack of willingness to learn modelling, it seems that modelling has an image problem, people are scared off and do not want to model, despite the views of some interviewees: *"it's fun to model."*

Another human factor is a lack of alignment between modelling practices of different developers. Different styles of modelling can lead to different subdivisions of models, and difficulties in understanding large models, if parts are created in different ways. This is a problem similar to traditional, code-based

software development, in which the use and enforcement of coding standards is well established. For this aspect, the main difference between the code and model-based industrial practice is tool support. Several interviewees remarked that the tooling lacks, e.g., checking of conformance of models to design guidelines.

Business

In addition to these human factors, impediments were mentioned that are related to the business perspective. Any change in process and tooling requires an investment, be it time, money, or both. The advantage of introducing these processes is often not easily quantified, making it difficult to gather support for them.

A difficulty in achieving a complete modelling pipeline can also be to get all involved disciplines in a company on board. A coupling between models from different domains is desired, but a lack of willingness and resources to do so hinders this synchronization between parts of a company. It might be that engineers estimate the amount of required resources as high, due to other functional or process impediments they see.

Further, engineers mentioned a lack of time and knowledge to invest in setting up a toolchain. Especially considering the need for customization of such toolchains, since almost no two companies are working with the same sets of tools. Furthermore, this customization depends on the type of product developed. *"It depends a lot on the domain what the generated code should exactly look like."* The domain furthermore impacts the required lifetime of products, complicating, as mentioned earlier, configuration management. Another related challenge that might apply is the need for the developed code to conform to strict regulations and certifications.

7.3.4 Future visions

Some alternative future visions were proposed, which would make this more useful and more possible. One of them describes an extension of existing tooling in to other modelling domains, such that all modelling activities, from system architecture to software implementation, can be performed in a single tool. An

alternative vision assumes that engineers from each domain will keep using their preferred tools, but aims rather at better interactions between these tools. Ultimately, both these visions allow for CI involving all models, by resolving one of the main seen impediments by practitioners: a lack of tool interoperability.

7.4 Discussion

The identified challenges in Section 7.3 paint part of the picture of the impediments towards adoption of CI in industrial MBD practice, by considering the different points of view from the different industrial partners. Still, the sample size is not big and we should be careful to generalize our findings to cases in which CI and MBD are adopted to different degrees than in the interviewed companies.

The division of the impediments in the different categories emphasizes the broadness of the encountered challenges. Indeed, a silver bullet does not exist, rather, it is a long process to introduce modelling and then CI in industry practice. While for our research, the functional and non-functional aspects are the most interesting to focus on, the human and business aspects should not be disregarded.

Considering the adoption of CI in modelling in the involved companies, it is noteworthy that in company 3, modelling and CI works well, using a single modelling language. Company 2 as well uses a single modelling tool for software models and develops them in rapid iterations. Tool interoperability and model interoperability is in these cases to a large extent avoided. Nevertheless, implementing a CI pipeline in which models are included for system design, detailed design, and implementation requires information sharing between these models and thus communication between the tools in which they are created. Indeed, these two elements are central to introducing CI in MBD and they make resolving the other named impediments more complicated. For example, impact analysis is more complicated when the impact must be assessed across modelling languages.

Considering these findings in another way, we can say that there are few impediments to introducing CI when modelling for code generation using a single modelling language. Rather, most of the found impediments are encountered

when models are also used in other parts of development, such as system design. Then, maintaining sound architectures and consistency between the models is increasingly challenging, particularly in a CI environment. An apparently practiced way to circumvent many of these impediments is by applying CI only to the lowest level models, those used for code generation, while manually managing the correctness of the other models. The downside of this way of working is that it benefits minimally from one of the main promises of CI, i.e., always having an overview of the state of integration, since that state cannot be completely known using only implementation models.

As also found, introducing CI is not always the desired approach, we have seen a current way of working in which introducing more frequent integrations is not useful. In that case, because there is a strict separation between design and implementation and because many manual steps are involved that would not scale up appropriately. It should be noted that the starting point of a company is crucial in how engineers view this aspect, conservative views are natural given a well-functioning process and foreseen serious impediments to changing them. Furthermore, if the benefits of their introduction are not clear, gathering support for CI practices is difficult. Finally, the domains in which the companies work and the traditions that those bring with them seem to impact the willingness to adopt faster development cycles.

7.5 Related work

Our study has underscored some results that were found earlier, when investigating the adoption of modelling practices in industry. In their experience report from 2005, Baker, Loh, and Weil [2] already note a lack of performance of tools and interoperability between different tools. Other empirical studies also point to tools as impeding more MBD adoption [4]. In addition to these technical issues, Hutchinson, Whittle, and Rouncefield [15], also using semi-structured interviews, find organizational (process and business-caused) factors important to the success or failure of their introduction. Recently, an evaluation of industrial MBD practice shows its potential benefits but also highlights the difficulties in its adoption, particularly tool interoperability and a steep learning curve of the method itself and tools used for it [5].

In a similar industrial context as our study, an interview study has shown the state of practice and impediments to introducing continuous deployment in agile software development projects [16]. While the paper does not consider modelling, it does identify some impediments that are also relevant in our case. Notably, in companies moving towards CI, one of the impediments is the complexity of test automation. This aspect is underexposed in our interviews, possibly because many of interviewees were doing system modelling and did not operate closer to the implementation.

Another interview study involving large industrial partners identifies four categories of impediments to introducing CI, albeit not in modelling projects [17]. The categories identified in that work are related to testing processes, the usability of tools, the splitting of the system into parts and the division of work among engineers. Some overlap can be noticed between those results and our results, particularly the impediments related to the process.

In our earlier work [18], we have reviewed modelling tools and their suitability to be applied in the context of CI and MBD. There, we concluded CI is achievable when using a single modelling tool, albeit possibly challenging to set up, but much more challenging when using a combination of modelling tools. This aligns with our findings in this work, where we have seen one company where a pipeline is implemented, but for a single tool. Further, we have seen in all companies that introducing or streamlining CI practices raises challenges in tool interoperability, and model interoperability, when using different modelling tools and modelling languages.

A systematic literature review in the area of combining agile methods (of which CI is one) and modelling has shown the immaturity of the field [19]. In particular, the authors have argued for more reports on industrial experiences. One such study focused on introducing agile practices in modelling projects in the automotive industry [10], although not including CI yet, the authors do report a successful application of agile methods. Similarly, a case study in the telecommunications domain has shown benefits of agile MBD [9], although it minimally discusses the extent to which this practice includes continuous integration. This paper contributes to this knowledge base by exploring views and current practices of industry practitioners of agile development methods in MBD projects.

Other work has reported on experiences of introducing CI in an MBD project [11]. The authors show the benefits from combining CI and MBD to the development process and discuss hurdles overcome to achieve this. An impediment identified in common with this work is the lack of support for differencing and merging of models.

Other work has considered practical applications of CI in the automotive domain [20]. While not explicitly about modelling, some identified impediments are closely related to our findings. In particular the many manual steps that are performed to move data in between tools, due to a lack of tool interoperability. Further, the authors identify many organizational issues, besides the tooling, which we have touched upon in the category of business impediments.

Although not evaluated in industrial practice, some initial works have sketched the possibilities to wrap modelling tools for each step in the process of modelling to validation in a continuous delivery pipeline [21]. This addresses the tool interoperability challenge for one specific set of tools. The authors stress the need for maturity of individual tools and steps in the pipeline and summarize the challenges of creating a CI pipeline for models by stating that all steps in it should be “model-aware.”

7.6 Conclusion

In this work, we have identified several impediments to introducing continuous integration (CI) in model-based development (MBD) through eleven interviews at three large companies. Furthermore, we have discussed how some of those impediments are circumvented in current practice. The three companies each have a different current way of working and therefore a different starting point when introducing or streamlining existing CI practices. Some starting points imply that introducing CI is not desired by engineers. Other starting points did see an embrace the idea of combining CI and MBD, but several impediments were named towards that goal. We have categorized these impediments as functional, non-functional, human, and business-related.

To broaden the coverage of our findings, in future extensions of this work, we aim to include more companies on different positions in the graph in Figure 7.3. For instance, by including companies that are doing more modelling

than company 2 and less CI than company 3. Moreover, we aim to include companies that have implemented more explicit links between all models.

To answer the research question posed in the introduction: the impediments to applying continuous integration in MBD projects are summarized in Table 7.1 and in addition also implicitly include general impediments to introducing MBD. Our two main findings of this interview study are 1) that introducing CI is not always desirable or useful given a current way of working, and 2) workarounds to common problems of tool interoperability and model synchronization are impeding the introduction of an automated CI pipeline for MBD.

Bibliography

[1] Douglas C Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[2] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context—Motorola Case Study. In *LNCS 3713*, pages 476–491. Springer, 2005.

[3] 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)*, pages 471–480. IEEE, 2011.

[4] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. 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, Feb 2013.

[5] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[6] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering (Innsbruck, Austria, 2013)*, pages 736–743, 2013.

[7] Ade Miller. A hundred days of continuous integration. In *Agile*, pages 289–293. IEEE, 2008.

[8] Martin Fowler and Matthew Foemmel. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[9] Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE software*, 28(2):84–91, 2011.

[10] Ulf Eliasson, Rogardt Heldal, Jonn Lantz, and Christian Berger. Agile model-driven engineering in mechatronic systems-an industrial case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 433–449. Springer, 2014.

[11] Vicente García-Díaz, Jordán Pascual Espada, Edward Rolando Núnez-Valdéz, G Pelayo, B Cristina Bustelo, and Juan Manuel Cueva Lovelle. Combining the continuous integration practice and the model-driven engineering approach. *Computing and Informatics*, 35(2):299–337, 2016.

[12] Hessa Alfraihi, Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Mohammadreza Sharbaf, and Howard Haughton. The Impact of Integrating Agile Software Development and Model-Driven Development: A Comparative Case Study. In *International Conference on System Analysis and Modeling*, pages 229–245. Springer, 2018.

[13] INCOSE. *Systems Engineering Handbook, v3.2.2*. 2011.

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

[15] 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.

[16] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the 'Stairway to Heaven'–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 392–399. IEEE, 2012.

[17] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Continuous integration impediments in large-scale industry projects. In *Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA)*, pages 169–178. IEEE, 2017.

[18] Robbert Jongeling, Jan Carlson, Antonio Cicchetti, and Federico Ciccozzi. Continuous integration support in modeling tools. In *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems)*, volume 2245, pages 268–276. CEUR-WS, 2018.

[19] Hessa Alfraihi and Kevin Lano. The integration of agile development and model driven development - a systematic literature review. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, pages 451–458. SCITEPRESS, 2017.

[20] Eric Knauss, Patrizio Pelliccione, Rogardt Heldal, Magnus Ågren, Sofia Hellman, and Daniel Maniette. Continuous integration beyond the team: a tooling perspective on challenges in the automotive industry. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 43. ACM, 2016.

[21] Jokin Garcia and Jordi Cabot. Stepwise adoption of continuous delivery in model-driven engineering. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 19–32. Springer, 2018.

Chapter 8

Paper C: Lightweight Consistency Checking for Agile Model-Based Development in Practice

Robbert Jongeling, Federico Ciccozzi, Antonio Cicchetti, Jan Carlson
In the Journal of Object Technology, Volume 18, no. 2 (July 2019).

Abstract

In model-based development projects, models at different abstraction levels capture different aspects of a software system, e.g., specification or design. Inconsistencies between these models can cause inefficient and incorrect development. A tool-based framework to assist developers creating and maintaining models conforming to different languages (i.e. *heterogeneous models*) and consistency between them is not only important but also much needed in practice. In this work, we focus on assisting developers bringing about multi-view consistency in the context of agile model-based development, through frequent, lightweight consistency checks across views and between heterogeneous models. The checks are lightweight in the sense that they are easy to create, edit, use and maintain, and since they find inconsistencies but do not attempt to automatically resolve them. With respect to ease of use, we explicitly separate the two main concerns in defining consistency checks, being (i) which modelling elements across heterogeneous models should be consistent with each other and (ii) what constitutes consistency between them. We assess the feasibility and illustrate the potential usefulness of our consistency checking approach, from an industrial agile model-based development point-of-view, through a proof-of-concept implementation on a sample project leveraging models expressed in SysML and Simulink. A continuous integration pipeline hosts the initial definition and subsequent execution of consistency checks, it is also the place where the user can view results of consistency checks and reconfigure them.

8.1 Introduction

The Model-Based Development (MBD) paradigm holds the promise of improving productivity of the development process by promoting models as core artifacts, particularly in early development phases, i.e., specification and design [1]. Further, models are also used for advanced development activities such as simulation and code generation. Besides, in industrial contexts, models as main project artifacts play an important role in documentation and communication between different development teams [2]. Models are becoming critical assets for development of industrial systems and software, not only within single projects but over several projects through model reuse. In modern industrial MBD practice, software systems are modelled through multiple views, using so-called *multi-view modelling* [3].

Views are represented by *heterogeneous* models, i.e., models conforming to different modelling languages (often created with different tools, which complicates consistency checking). Usually, these views are exploited by different teams and for different aspects of development. Consider the context shown in Figure 8.1, where a system model, created by system designers to describe

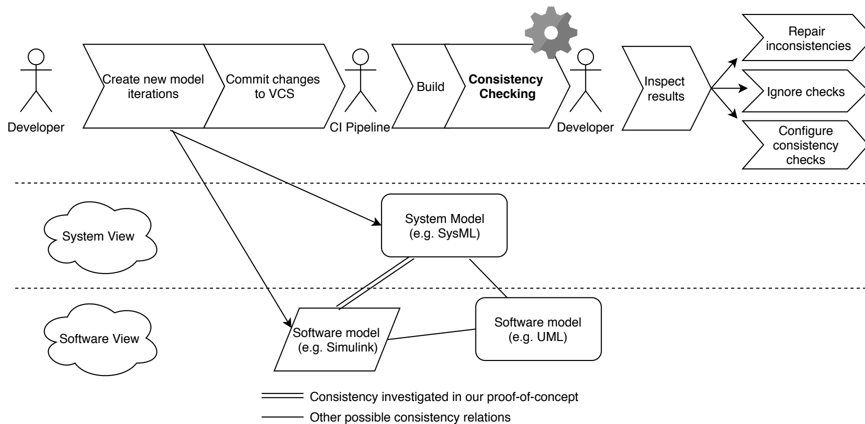


Figure 8.1. Illustrating the scope of our approach, a tool performing inter-model consistency checking between heterogeneous models and notifying the developer about inconsistencies.

architectural matters, is refined into a set of software models by the software designers. In many cases, models across different views are closely related and they may partially overlap since they describe the same parts of a system. The use of multiple (often partially overlapping) views requires a careful checking and maintenance of consistency among them. Consistent models are in fact essential to ensure a coherent design as well as efficiency and correctness in the development process. While complete consistency (at any time in the development) may not be achievable or desirable, lingering inconsistencies can snowball into serious issues if not identified in early phases of development. A way to prevent this is to notify the developer about inconsistencies between models soon after their introduction, by means of consistency checking.

Consistency checking within a model (i.e. intra-model consistency), or between models conforming to the same modelling language, is often available in modelling tools. We focus on checking inter-model consistency between heterogeneous models, which is a more complex endeavour for several reasons. Firstly, inter-model consistency often requires the ability to interact with a set of different modelling tools and processes in an industrial MBD context. Changes in this ecosystem are hard to make. Replacing a modelling tool to be able to perform inter-model consistency checks is often not feasible and any additional tool should not interfere with the existing ecosystem. Similarly, existing development processes are not easily changed, additional actions would be performed reluctantly in the best case, or skipped in the worst case, if they disrupt existing processes. Secondly, when consistency checks require a steep learning curve or excessive effort to create or maintain, the intended users may be discouraged from using them in the first place. Existing approaches, e.g. those based on Triple Graph Grammars [4] or link-models [5], are powerful but complex, hence requiring considerable effort to define and maintain consistency checks.

The context of this research is represented by an academia-industry collaboration called Software Center[1] and composed of 12 large companies and 5 universities. Among our industrial partners in Software Center, there is a clear trend of model-based development going agile. Very short cycles, typical of agile development, complicate consistency checking, especially if it requires a

[1] www.software-center.se

large effort in defining, maintaining and executing consistency checks. A consequence is that there is a need for a lightweight consistency checking approach. Lightweight means that it shall infringe minimally on existing development processes and tools, but aid developers in easily monitoring inter-model consistency. This kind of approach is currently lacking and much desired by practitioners.

In this paper, we show an application of consistency checking between heterogeneous models. We motivate requirements for a lightweight approach in Section 8.2, present a generic approach that satisfies these requirements in Section 8.3, and show an implementation of this approach in Section 8.4. Limitations and potential extensions to our approach are discussed in Section 8.5, a relevant portion of the extensive related work about consistency management is discussed in Section 8.6, while conclusions and some prospects of future work are included in Section 8.7.

8.2 Scope

We have already introduced the need for lightweight consistency checking. This section describes further our target industrial MBD context. From it, we derive a set of requirements for a lightweight consistency checking approach that is useful and usable in practice.

8.2.1 Industrial context of consistency checking

Multi-view modelling refers to a practice in which a system is designed using multiple models (each of which representing a specific modelling view), potentially created in different tools and described by means of different languages [6]. Different models may describe the system under development, or just part of it, at different levels of abstraction and from different stakeholder perspectives, such as requirements engineer, system designer, or software developer. Yet, these models are commonly not disjoint, since they describe (parts of) the same system. There is often an explicit overlap, where multiple models describe, in the same or different levels of detail, the same parts of the system.

Kolovos et al. [7] classify the relationships between models that induce this overlap, of these, the most relevant in industrial practice are “uses”, “refines”,

"complements", "alternative for", and "aspect of". Due to the nature of these relationships, they are highly correlated to the structure of the models. Kolovos et al. [7] go on to classify types of inconsistencies that can occur between overlapping models, the ones relevant to us are "incompleteness", "contradiction", "misuse", and "redundancy." Intuitively, a comparison of the structure of two overlapping heterogeneous models would show these types of inconsistency at a glance. While these relationships can occur between any pair of models, in our industrial context, we are primarily interested in consistency between models across different levels of abstraction, i.e., vertical inter-model consistency [8].

For example, let us consider a system model containing a SysML block B with two ports, P_1 and P_2. During system specification, the system designer might model parts of the system as a "black box", i.e., stop modelling at this level of abstraction and only care about the interfaces between blocks. Software designers on the other hand, as part of the system design, would model this as a "white-box", down to a more detailed level. They might, for instance, create a Simulink model S that describes B in more detail, with input and output ports corresponding to P_1 and P_2, and with additional details not included in B. This type of view relation between models S and B is commonly called *refinement* from S to B, or *abstraction* from B to S, respectively [9]. Other examples of these refinement relations include the one between a SysML model and an EPLAN[2] model to capture hydraulic schematics and between a SysML model and a Modelica model capturing the control system and dynamic behaviour, as exemplified in [10].

Figure 8.1 shows an overview of an industrial MBD context for which our proposed consistency checking is intended. Model inconsistencies across views, and thereby across e.g. specification, design, and implementation, complicate the development and evolution of systems. Inconsistencies shall never uncontrollably spread through the system design and one way to avoid this issue is by introducing consistency checks to support developers in identifying, at an early stage, possible inconsistencies in the system under development. Therefore, as shown in Figure 8.1, the development team is aided, during development and evolution of the architectural and software models, in keeping these models consistent through lightweight checks that indicate discrepancies in the structures

[2]https://www.eplanusa.com/us/home/

of the created models. Note that in the different views, several heterogeneous models could exist, for example UML models in the software view (as shown in Figure 8.1). We highlight the generic applicability of our approach by choosing different languages in the example shown in Section 8.4.

To summarize, usable consistency checking, to ensure that models express overlapping concepts from different point of views without contradicting each other [11], is pivotal for multi-view modelling approaches to be efficient. For industrial adoption, tool support is vital, too. Next, we elaborate on which requirements an industrial application of such a consistency checking mechanism entails.

8.2.2 Requirements

Since models conforming to different languages are typically designed using different tools and ensuring consistency is often a manual task, inconsistencies between them could remain unnoticed for considerable time during development. This is particularly true when models are created in different views and for different aspects of the development. Let us exemplify in the context shown in Figure 8.1. During the specification and design of a car, a system model denotes the overall design of the car and more detailed models are designed to describe software, electronics, braking system, etc. A possible inconsistency could be introduced between the structural model, conforming to SysML, and the refining functional model, conforming to Simulink, that fails to refine a particular block of interest as defined in the structural model. We aim to support the checking of vertical consistency between heterogeneous models in cases where models are related by one of the aforementioned relations and a certain overlap in the structure of the models exists. As already mentioned, notifying developers of possible inconsistencies of this type is considered as very helpful in industrial practice, given the complexity of the systems and the distribution of the development efforts.

Overlaps causing possible inconsistencies are, in most cases, not one-to-one relations between entire models, nor between model elements at the same granularity level. Rather, since different models describe the same parts of the system at different levels of abstraction, the overlap is more likely to spread across the different levels of granularity, e.g. an entire model refining a subsystem,

or a package of multiple blocks refining a model. For example, in the case of SysML and Simulink models describing the same system, a Simulink subsystem might not map one-to-one to a SysML block, but rather the SysML block might be refined via an entire Simulink model, containing several subsystems. Our approach allows the definition of consistency checks between related model elements across different languages and granularity levels. Since model elements may represent complex sub-models (a model element being the container root of a sub-tree of contained model elements), our approach should be able to recursively execute consistency checks too, to account for hierarchical compositions and containments across models.

The need for consistency checking becomes more pressing when companies adopt agile multi-view modelling, in particular, when the development includes continuous integration (CI). CI refers to the practice in which developers integrate their work frequently, multiple times per day, in a shared repository [12]. In this context, inconsistencies between heterogeneous models are easy to overlook but nevertheless important to identify as soon as possible, to prevent them from rapidly spreading to related artefacts. Agile development implies that models are developed in short iterations and in parallel with other models. Consequently, any of the overlapping models can be seen as anticipating changes in the others at any time during development, e.g., the system model may not yet contain concepts already described in software models and vice versa. In these settings, inconsistencies are inevitable and almost required, since forbidding them would hinder the concurrent and incremental nature of agile. Automatic resolution is undesirable too, since in most cases it can not be determined which of the involved models should be reconciled in a scenario where any can be anticipating the others. Moreover, temporary inconsistencies are sometimes required to allow for particular development activities [13]. Therefore, we want to allow developers to choose if and how to act on detected inconsistencies. For this reason, we propose a consistency checking approach that identifies and indicates inconsistencies to the developers, without enforcing their resolution.

The frequency by which inconsistencies are presented to the developer, if not on-demand, is a sensitive matter: if too frequent, it becomes annoying, if too seldom, it becomes irrelevant. The CI pipeline provides a middle-ground, where inconsistencies can and should be presented at the time of pushing changes to

the shared repository. Furthermore, it provides an environment independent of any particular modelling tool, where to configure consistency checks and view their results.

Industrial MBD practice typically involves many different tools, modelling languages, and development processes. Often, techniques fail because the process view is not taken into account. For example, because for the introduction of consistency checks, large changes to this environment, or to existing development process, are undesirable. Therefore, our approach should have a small footprint, i.e., be a minimal addition to existing MBD environment and a minimal added effort in existing development processes and ways of working. We aim for the application of consistency checks in an agile MBD process and in particular in a CI pipeline, so we must also minimize their interference with the developer flow. Consistency checks should thus also be lightweight with respect to the required effort to create, maintain, and use them. The checks themselves should be frequently executed, applicable to multiple languages and allowing for checking consistency across granularity and abstraction levels.

Table 8.1 summarizes the requirements described in this section and their motivation. Our goal is to provide an approach, and tool support, for detection and notification of inter-model inconsistencies, across heterogeneous models and in a CI pipeline for agile MBD projects. We focus on structural equivalence between model elements or parts of models, as well as for structural refinement between model elements and parts of models.

8.3 Our consistency-checking approach

In this section we outline the constituents of our approach for checking consistency between models expressed in different views and languages.

The types of consistency interesting for the developer depend on the involved modelling languages and the system under development. Hence, the meaning of consistency cannot be decided a priori, but should rather be specified by the person defining the consistency checks. In some existing consistency checking approaches, the meaning of consistency is captured in an intermediate translation, like a case by case dictionary, formally defining how to compare model elements between different models (and languages). An example of fixed medium to

Table 8.1. Industrial practice (left) and the corresponding requirements (with ID) they entail (right).

During development, models are:	So, consistency checks should:
Created in different languages and tools.	R1. Check inter-language consistency.
Partly overlapping.	R2. Compare the structure of models.
Related by refinement or equivalence at between model elements.	R3. Allow consistency definition across model elements at different granularity.
Purposefully, temporarily, inconsistent.	R4. Not attempt automatic resolution.
Changed continuously.	R5. Be executed frequently.
Created in complex environments.	R6. Have a minimal impact to the existing environment.
Created in complex processes.	R7. Be easy to create, use and maintain.

express these ‘dictionary entries’ is Triple Graph Grammars (TGGs) [4]; this and other related mechanisms are discussed in more detail in Section 8.6.

In these approaches, each dictionary entry (mapping) describes two types of information. The first maps meta-model elements between different languages and how to check consistency between them. The second denotes model elements, across heterogeneous models, between which consistency should be checked. The user is expected to define both for each entry. In our approach, we propose to simplify the task of creating these entries by splitting the two information types as follows.

Mappings between meta-model elements across different languages and the definition of the various kinds of consistency that can be checked (e.g., name equivalence) are described in *language*[3] *consistency mappings*. Mappings of model elements, across heterogeneous models, between which consistency

[3]Note that in the paper we use ‘language’ and ‘modelling language’ interchangeably as synonyms.

should be checked and which specific kind of consistency to check are described in *model consistency mappings*. The user is only concerned with declaring and maintaining model consistency mappings, while the labor invested in creating language consistency mappings is limited to a one-time effort, unless the language undergoes changes. This makes the usage of our approach lightweight. Since we are dealing with heterogeneous models, in order to be able to compare them, and thereby check consistency, we need to represent them in a common notation.

A consistency check CC is composed of one language consistency mapping LC_{map} and one model consistency mapping MC_{map}. The remainder of this section presents LC_{map} and MC_{map} in detail and shows an overview of all the steps required for the definition and execution of consistency checks.

8.3.1 Language consistency mapping

A language consistency mapping LC_{map} consists of:

(1) a relation between different languages (at meta-element level), and

(2) the definition of consistency types.

As mentioned before, we aim at checking consistency by comparing models structure and their hierarchical nature. To structurally compare two heterogeneous models, we need to bring them to a common notation that highlights their structure. We opted for a tree-based notation since it permits to capture structures, and hierarchies, in a convenient and compact way. Furthermore, it is generic enough to represent models conforming to, potentially, any modelling language that entails structural modelling in a hierarchical fashion. Since we address in this case specifically comparisons of model structures, a tree structure suffices. In more general cases, more generic structures would be more appropriate. A tree is an abstract representation of a non-empty set of model elements, which precisely reproduces the model hierarchical structure. Model elements become nodes.

For example, in the case of Simulink models, blocks, subsystems and ports can be mapped to tree nodes, together with their hierarchical structure, whereas the operations inside blocks are not. Figure 8.2 shows an example of tree

representation of a Simulink model, where "distiller" contains a subsystem "Distiller", which in turn contains subsystems "Heat_Exchanger" and "Boiler", which are in that hierarchy mapped to nodes in the tree.

Nodes inherit names from respective model elements and they are assigned an abstract type for comparison purposes (e.g., a Simulink inport and a SysML flowport become nodes of type 'port'). Types can be leveraged to check consistency in cases where name equivalence does not hold. For example, to check that two blocks, one in a SysML model and another in a Simulink model, contain the same number of 'ports', regardless of the names of these blocks and ports.

A LC_{map} between language L_A and language L_B consists of:

(1) two separate transpositions, from L_A and L_B to a tree-based notation TN, and

(2) a set of comparison rules between L_A and L_B (e.g. name equivalence) done at the TN level.

Figure 8.3 shows how a LC_{map} is used for comparing models. Technically, two models M_A conforming to L_A, and M_B conforming to L_B, are transposed into two corresponding trees T_A and T_B, conforming to TN, and comparisons are done between T_A and T_B.

Our proof-of-concept implementation provides two comparison rules, one for equivalence and one for refinement, exemplified in Figure 8.4. Since we can do comparison based on node names, types, and structure of their tree representations, we defined three levels of consistency *strictness*:

- Strict: when comparisons are based on node names, types and structure;
- Intermediate: when comparisons are based on node types and structure;
- Loose: when comparisons are based on structure only.

Equivalence between nodes $n_A \in T_A$ and $n_B \in T_B$ is defined as follows, with respect to the strictness levels:

- Strict: n_A and n_B have the same name and type and the same number of children; in addition, each child of n_A has a *strict* equivalence to a child in n_B and vice versa;

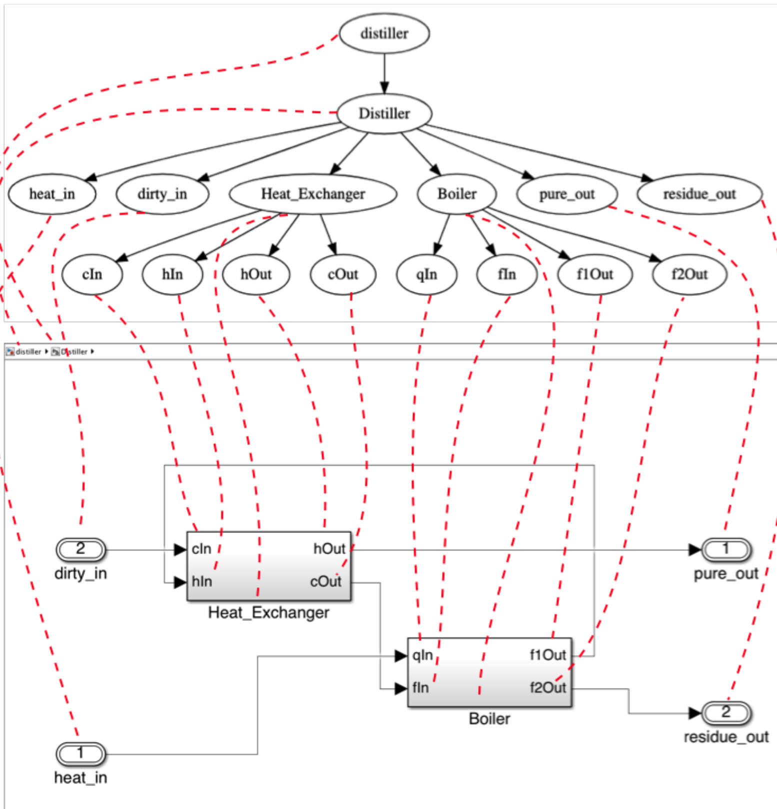


Figure 8.2. Example of a transposition of a Simulink model to tree. Subsystems and ports are mapped to nodes in the tree, but not the simulation blocks inside the subsystems. The Simulink model is inspired by the well-known SysML Distiller example model [14].

- Intermediate: n_A and n_B have the same type, the same number of children; in addition each child of n_A has an *intermediate* equivalence to a child in n_B and vice versa;

- Loose: n_A and n_B have the same number of children; in addition, each child of n_A has a *loose* equivalence to a child in n_B and vice versa.

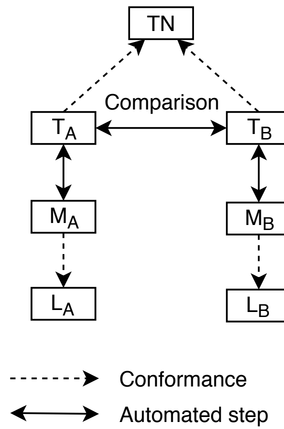


Figure 8.3. LC_{map} consists of separate transpositions from both languages to a tree-based notation and a number of comparison rules. When executing a consistency check, automated model transformations transpose models into trees, between which automated comparison is run.

Refinement between nodes $n_A \in T_A$ and $n_B \in T_B$, where n_B refines n_A, is a directed relation defined as follows, with respect to the strictness levels:

- Strict: n_B has at least the same number of children of n_A and each child of n_A has a *strict* equivalence to a child in n_B.

- Intermediate: n_B has at least the same number of children of n_A and each child of n_A has an *intermediate* equivalence to a child in n_B.

Note that we do not define loose refinement, since its checking would not lead to meaningful inconsistencies.

Comparison rules – equivalence or refinement – can be defined between any pair of nodes $n_A \in T_A$ and $n_B \in T_B$, also when placed at different hierarchical levels in the respective trees. For two trees to be consistent (either through equivalence or refinement), their root nodes should be consistent. This also means that, if comparison rules are defined between roots of sub-trees, then all nodes above them would not be considered for consistency checking.

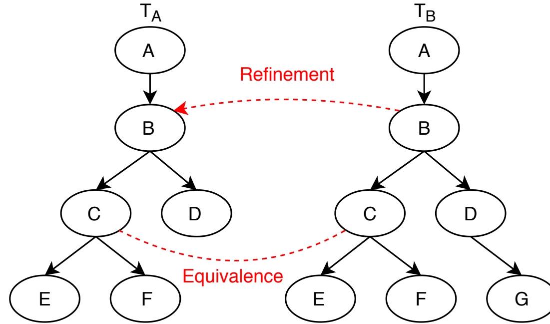


Figure 8.4. Examples of an equivalence relation and a refinement relation between T_A and T_B. Node C in T_A is strictly equivalent to node C in T_B and node B in T_B strictly refines B in T_A.

8.3.2 Model consistency mapping

A consistency check CC requires, in addition to a LC_{map}, a model consistency mapping MC_{map}, which consists of:

- two model elements, between which consistency should be checked,
- the type of consistency to check, and
- the level of consistency strictness.

To define MC_{map}, the user only needs to configure these three parameters. Automated mechanisms implementing LC_{map}, and the comparison rules defined in it, are then responsible for generating and executing the consistency checks. Once defined, CC can be executed at any time throughout the evolution of the entailed models, with the possibility to adjust its configuration if needed. Future extensions of our approach will reduce the effort of defining consistency checks by automated support, for instance by suggesting model consistency mappings based on potential matches identified through a similarity analysis between the heterogeneous models to be compared.

As mentioned in the explanation of language consistency mappings, the tree-based notation allows to easily compare models and their elements, also

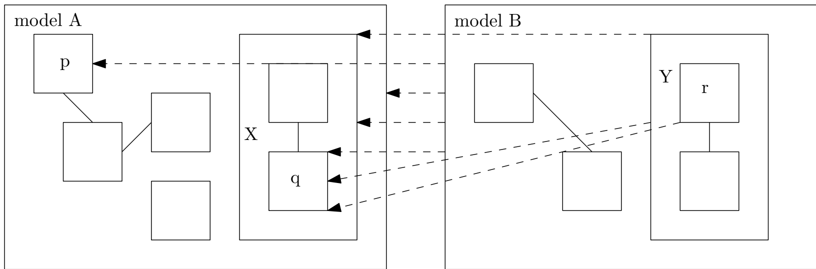


Figure 8.5. Example of possible model consistency mappings between an abstract model A, refined by a model B. The dashed lines indicate possible refinement relations at different granularity levels.

when placed at different hierarchical levels. Figure 8.5 illustrates examples of model consistency mappings. For instance, model B could be a refinement of model A, or parts of it, such as sub-model X or elements p or q. Similarly, parts of model B, for example sub-model Y, could refine sub-model X or element q. Lower level mappings are possible too: for example element r in model B refining element q in model A.

These model consistency mappings relate two model elements, but can be used to check consistency between more than two model elements, by chaining consistency checks. For example, to check that elements a, b, and c are equivalent, two consistency checks can be defined, one checking that a is equivalent to b and the other checking that b is equivalent to c. Future extensions of our approach will support grouping these checks such that one result summarizes all of them. For instance, if in the above example a is equivalent to b but b not to c, the grouped check would fail too.

8.3.3 Continuous integration pipeline

The execution of consistency checks is embedded in the CI pipeline, triggered by a model change that is pushed to a common repository, and executed after a build. A high-level description of the execution and configuration of a CC consists of the following steps:

1. MC_{map} is evaluated. Consider a mapping between model element e_A

of model M_A in language L_A and a model element e_B of model M_B in language L_B

(a) LC_{map} between L_A and L_B is used to create trees T_A and T_B from models M_A and M_B, respectively.

(b) In T_A and T_B, nodes corresponding to e_A and e_B are compared using a comparison rule, corresponding to a combination of the type of check (equivalence or refinement) and the strictness level (strict, intermediate, or loose). Since comparison rules define a comparison between nodes by including, recursively, their children, technically the subtrees with root nodes represented by e_A and e_B are compared.

(c) The result of executing the CC is summarized as a binary outcome: pass or fail. In case of a failed check, a summary of the reasons behind the failure is shown to the user.

2. Configuration of existing model consistency mappings can be modified, including options to mute or skip checks in future runs.

3. The user can also add or delete model consistency mappings.

8.4 Proof of concept

In this section we present a proof-of-concept implementation[4] of our approach. The approach is implemented as a plug-in for Jenkins[5], a tool supporting automation of CI pipelines. In such a pipeline where a CI server is already in place and used to monitor the state of the development, including our consistency checks in both the process and toolset requires only a minimal overhead.

In the remainder of this section, we show the process of defining and executing consistency checks on the Distiller example [14] and applying it to one model consistency need that we identified exists in our industrial partners: a

[4] For the interested reader, the implementation is available at: https://github.com/RobbertJongeling/consistency-plugin. A demo video (https://github.com/RobbertJongeling/consistency-plugin/blob/master/Demo.mp4) showing the approach at work is available in the GitHub repository too.

[5] https://jenkins.io/

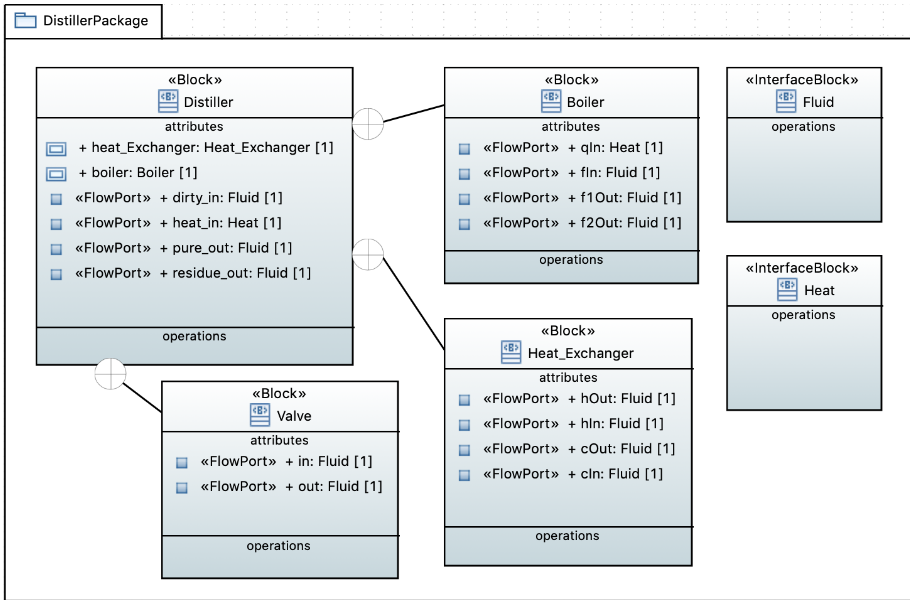


Figure 8.6. Simplified block definition diagram of the SysML distiller example [14].

functional Simulink model refining a structural SysML model. Two models are created, one SysML model, shown in Figure 8.6, and one Simulink model which refines selected subsystems of the SysML model, and which was shown earlier in Figure 8.2.

Next, we briefly present the LC_{map} between the two entailed modelling languages and illustrate the implemented plug-in at work through an example of a consistency check definition and execution.

8.4.1 Language consistency mapping

To transform the structure of Simulink and SysML models to trees, we defined two model transformations, which map concepts from the respective modelling languages to a tree-based notation. Both transformations are implemented

in Xtend[6], to allow seamless integration with the Java implementation of the Jenkins plug-in. Transformations take in input the model files as they persist in the file system rather than requiring multiple interfacing with modelling tools. The models are then parsed and model elements of interest, as defined in the language consistency mappings, are added as nodes to a tree. In the current implementation, LC_{map} is embodied in the model to tree transformations. We are currently working on a more flexible implementation, where we will separate the definition of LC_{map} from the model transformation implementation. Once LC_{map} is defined, a set of higher-order model transformations will generate specific model to tree transformations based on LC_{map}.

SysML. A subset of SysML diagrams is represented by structural diagrams, i.e., block definition diagrams and internal block diagrams. In this work, we focus on SysML models described in terms of these diagrams. In our tree-based notation, the root node represents the entire SysML model and the tree hierarchy reflects the structural hierarchy of the model. The root's children are packages or blocks. Packages can contain other packages and blocks, while blocks can contain other blocks and ports. The SysML model in the running example was created using Eclipse Papyrus[7]. The translation of the model to a tree is performed taking in input the *.uml* file, which contains the model definition (without diagrammatic information). In our transformations, we leverage the *EMF Ecore Resource* facilities to programmatically access the contents of this type of file.

Simulink. To parse Simulink models, from binary *.slx* or serialized *.mdl* format, we rely on CQSE's Simulink Library for Java[8]. As for the SysML model, the root node of the tree represents the Simulink model. The children nodes are then the SubSystems, Inports, and Outports contained in the models. SubSystems can contain other SubSystem, Inports and Outports. Note that we choose to omit certain types of blocks used to specifically implement Simulink

[6]https://www.eclipse.org/xtend/
[7]https://www.eclipse.org/papyrus/
[8]https://www.cqse.eu/en/products/simulink-library-for-java/overview/

simulations, such as logic operations and data conversions, since they do not affect the model structure.

8.4.2 A consistency checking tool

In this section we detail the approach steps enumerated in Section 8.3.

Defining model consistency mappings. Model consistency mappings are defined inside the Jenkins plug-in, by selecting the model elements between which consistency should be checked as well as the type and strictness of those checks. Figure 8.7 shows an example of consistency check definition. In our example, the type of model can be Simulink or SysML, but this can be extended to any language for which a transformation to the tree-based notation is implemented. When a modelling language is selected, the next drop-down box is populated with all model files of that language in the Jenkins workspace. When a file is selected, the next drop-down box is populated with all fully qualified names (FQNs) of model elements in the model, as represented in the related tree. Eventually, the strictness and type of check are selected. Note that, after checks are executed, the user can select to mute or skip them in future runs. Before executing the check, its result is set to *NYE* (Not Yet Executed), and no further comments are available.

Post-build: run consistency checks. We have implemented the execution of our consistency checks as a post-build action in Jenkins. After the build step, the execution of the consistency checks is triggered and results are shown.

Comparing trees. The first step in executing a consistency check is to transform the models to trees. Resulting trees for our running example are shown in Figure 8.8, where the black nodes represent the model elements to be compared (their selection in the MC_{map} can be seen in Figure 8.7. The refinement relation is now checked, not between the complete trees, but between the subtrees starting at the black nodes.

Consistency Check

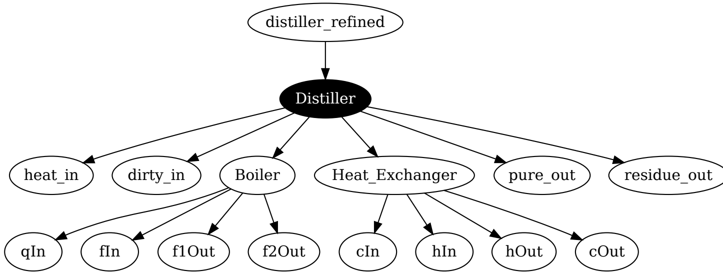
Result:	NYE
Comments:	
Type of model A	Simulink
File of A	simulink/distiller_refined.slx
FQN of A	distiller_refined/Distiller
Type of model B	SysML
File of B	sysml/DistillerExample.uml
FQN of B	DistillerExample/DistillerPackage/Distiller
Strictness	STRICT
Type of check	REFINEMENT

- [ ] Mute this check until a change
- [ ] Skip this check until turned on

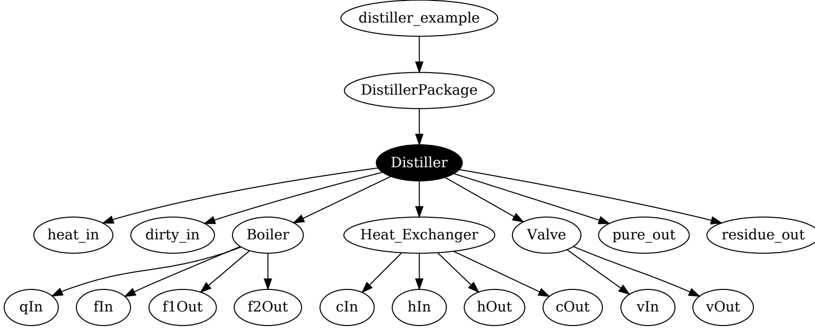
Delete

Figure 8.7. Example definition of a model consistency mapping in the Jenkins plug-in. Here, the element Distiller of the Simulink model distiller_refined is said to strictly refine the element Distiller in package DistillerPackage in the SysML DistillerExample model.

View results and manage configuration. In this case, the consistency check fails, since the model element in model A is not a refinement of the model element in model B. The *Valve* is in fact missing in the Simulink model as compared to the SysML model. This short explanation is shown in the result field of the MC_{map} definition, as shown in Figure 8.9. More detailed logs are available in the console output in Jenkins. A whole cycle of definition and execution of consistency checks is now completed. New consistency checks can be defined and existing ones edited or deleted. Model consistency mappings can also be left unaltered to be run again in future builds, or set to be skipped or muted. When a check is skipped, it is not executed in future builds, until the user enables it again. When muted, a check is executed but its results are hidden, unless they are different to previous results in its previous execution. This allows



(a) Tree representation of the Simulink model, only including *SubSystems* and *Ports*.



(b) Tree representation of the SysML model, only including Blocks and Ports.

Figure 8.8. Tree representations of the Simulink and SysML models; the subtrees with root nodes indicated in black are compared.

Figure 8.9. Result message of the failed strict refinement check.

the user to mute reports on inconsistencies that are relevant but temporarily tolerated, for example when modifying a model and before propagating the changes to other related models.

8.5 Discussion

In this work, we have focused on lightweight consistency checking to help developers discover structural inconsistencies between heterogeneous models. In particular, we have considered the requirements (Rx) summarized in Table 8.1. R1-R2-R3 are satisfied by choosing to construct an abstract tree representation from models. Indeed, this allows checking between models in different languages, since we compare their representations in a common format, but more importantly, this format represents the structural characteristics of the models, enabling their comparison. Comparison rules are defined between tree nodes, regardless of their position in the tree, so they enable consistency checking between model elements at different levels of granularity, for example an entire Simulink model can be compared to a single SysML block. R4 is fulfilled by providing detailed feedback on detected inconsistencies to the user, but not automatically resolving inconsistencies. R5-R6, regarding frequent execution and minimal impact on the existing ecosystem of consistency checks, are satisfied by the implementation of our approach in a CI pipeline. This provides a natural environment for executing the defined checks frequently, while not requiring a particular modelling tool nor notable changes to the development process. R7 states that our approach should consist of consistency checks that are easy to create, use, and maintain. This is satisfied by separating language consistency mappings from model consistency mappings, requiring the user to only input a small amount of information to generate and execute consistency checks. These checks are defined once and executed at each integration, unless

they are skipped, muted, or deleted by the user.

Evidently, the proof-of-concept implementation only focuses on a limited industrial context characterized by multi-view modelling and consistency checking, but we have argued its applicability in broader context. We exemplify our approach by applying it to check consistency between a SysML and a Simulink model, but the approach is generic enough to deal with many different situations from industrial practice. For example, to check consistency between EAST-ADL models and AUTOSAR models, UML models and Modellica models, or even between architectural models and code. One of the powers of our approach and implementation is that it can be easily extended to accommodate such checks, requiring few extra things than a language consistency mapping for those languages.

Applying the approach in those different scenarios requires generalizing it beyond its main limitation, i.e., its entailed type of only structural consistency. Such generalizations can be supported by opting for a different intermediate notation than the current tree structure. When we consider a different metamodel in this place, also the comparison algorithms can be extended to detect more different types of inconsistencies. For example, when we consider not just the structure of models but also values of variables, the intermediate notation should also contain this information and then a comparison algorithm can be devised that utilizes that information for inconsistency detection.

A smaller limitation, intrinsic to our approach, is a decreased level of control over the case-by-case semantics of consistency checks. Instead, this has been for ease of use: the user relies on a global language consistency mapping created once and only specifies for each consistency check in a minimal way what elements are to be checked for consistency and what type of consistency should exist between them. The latter definition is reused throughout the evolution of the models, the consistency check is executed whenever the models are changed. The very limited effort required to use it together with the relevance of the entailed spectrum of identifiable inconsistencies and its non-disruptive nature, with regards to the development process to which it is applied, make our approach promising for use in industrial contexts.

In the current implementation, we have focused on a specific example relevant to industrial practice. To perform a full-scale industrial evaluation

however, requires the implementation to be enhanced with additional language consistency mappings and capabilities to check other types of inconsistencies.

8.6 Related work

Consistency among and within views is pivotal to ensure efficiency and correctness in the development process [15]. This work provides an approach to lightweight consistency checking between heterogeneous models in a multi-view modelling context. In particular, we study an industrial multi-view modelling environment [2] in combination with agile development practices.

Dajsuren et al. [16], also consider consistency between different views. Similarly to our approach, the authors prototype a tool for SysML structural diagrams aimed at the automotive industry, but the underlying approach is applicable to other languages as well. To enable comparison between models, both are first expressed at the same level of abstraction. The resulting models are compared as graphs to detect inconsistencies based on missing model elements or relations in one model that are declared in the other model. In their approach, model elements are annotated directly in the modelling tool to denote consistency between model elements at the same granularity level. Similarly, our approach aims to compare consistency between two different views with some structural overlap, but in addition it allows for checks across heterogeneous models and model elements at different granularity levels.

In this work, we create an abstract tree representation of models to enable comparisons between them. Other works employ other formalisms to achieve the same goal of being able to compare models in different languages. An often used mechanism is Triple Graph Grammars (TGGs), which allow a formal definition of the mapping of model concepts across different languages [4], for instance between SysML and Modelica as done by Johnson et al. [17]. As opposed to our approach, these approaches require a high effort in declaring and maintaining the consistency checks.

The consistency checking approach proposed by Egyed allows for the creation of consistency rules in any formalism [18]. Notably, in this approach, consistency checks are only executed when model elements they cover are changed, thus improving over approaches in which batches of checks are exe-

cuted periodically. It can be a valuable future enhancement of our approach to similarly only execute those checks that relate model elements that have changed since the last execution.

Another means capturing the specific way of comparing particular model elements are link-models [5]. These link-models declare the relation between parts of models, and constraints on that relation, by relating model elements through particular types of links, equivalence, refinement or satisfies. The link-models are then used to derive validation rules that can be automatically executed. The applicability of this approach is limited to MOF-based models, whereas our approach is meta-metamodel independent.

Similar to our approach, also graph structures have been proposed as an intermediate representation of models as well as the starting point for detecting inconsistencies [19]. There, the graphs represent logical facts contained in the model, such that inconsistencies between graphs mean inconsistencies in the models. In our approach, the tree denotes not such logical facts, but rather focuses on the model structure.

In addition to approaches based on intermediate representations of models, others have proposed different means of comparison between models. For example, by declaring statements based on first-order logic to express facts that should be true about models [20]. Later, these ideas were more matured and generalized, for example in the Epsilon Object Language [21]. The advantage of our approach relative to these approaches is that the developer is not tasked with declaring such statements, since the meaning of consistency is captured in the language consistency mapping and the developer just specifies which model elements should be consistent.

The existing literature on consistency management in general is extensive [3], so necessarily, the included works cover only a small portion of it. Notably, Feldmann et al. categorize existing approaches as proof theory-based, rule-based, or synchronization-based [22]. Our approach can be categorized as synchronization-based, where the language consistency mappings define how model elements should be compared between languages, albeit not by a direct comparison but through an intermediary tree structure. Moreover, a plethora of approaches exists for consistency checking between UML models [23]. Even though there are numerous approaches presented, we are not aware of any

approach satisfying the requirements with respect to lightweightness as listed in Section 8.2.

8.7 Conclusions and future work

In this work, we argued for inter-model consistency checks that are lightweight, i.e. easy to use and non-intrusive as they identify inconsistencies but do not strictly enforce consistency. The creation and maintenance of consistency checks is simplified by separating their definition in a globally reusable part, the language consistency mapping, and a simple specific definition, the model consistency mapping. The model consistency mapping can be used to notify the user throughout the (possibly parallel) evolution of involved models. We provided a proof of concept implementation and showed how the approach works on a simple example of inter-model consistency between models conforming to different languages.

While our approach is applicable to MBD in general, we showed its feasibility in agile MBD settings, by leveraging CI and related tools to implement consistency checks. Through our proof-of-concept, we showed the ease by which a user can define checks at different granularity levels and between heterogeneous models. Moreover, we showed the usefulness of lightweight checks for inter-model consistency in a CI pipeline, as well as possible interactions between a CI server, modelling tools and version control systems. In agile MBD settings, this approach allows simple explicit checking of consistency between a large number of model elements, thereby highlighting at a glance, and soon after their introduction, structural inconsistencies that may be costly to fix if detected at a later stage.

In our future work we plan to build upon the approach presented in this paper to enable the detection of additional and more complex inter-model inconsistencies, while maintaining its lightweight nature. Moreover, we will provide features to further simplify the manual definition of model consistency mappings, e.g. by having the tool to automatically suggest likely candidates. An evaluation of our approach in terms of an industrial case-study or controlled experiment will follow once the implementation will be more mature (and including the future enhancements listed in this paper).

Bibliography

[1] Douglas C Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[2] Jagadish Suryadevara and Saurabh Tiwari. Adopting MBSE in Construction Equipment Industry: An Experience Report. In *25th Asia-Pacific Software Engineering Conference APSEC*, 2018.

[3] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software & Systems Modeling*, pages 1–27, 2019.

[4] Hartmut Ehrig, Karsten Ehrig, and Frank Hermann. From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. *Electronic Communications of the EASST*, 10, 2008.

[5] Stefan Feldmann, Manuel Wimmer, Konstantin Kernschmidt, and Birgit Vogel-Heuser. A Comprehensive Approach for Managing Inter-Model Inconsistencies in Automated Production Systems Engineering. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1120–1127. IEEE, 2016.

[6] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, Sep 2017.

[7] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *2008 1st In-*

ternational Conference on Software Testing, Verification, and Validation, pages 356–364. IEEE, 2008.

[8] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. Consistency Problems in UML-Based Software Development. In *UML Modeling Languages and Applications*, pages 1–12. Springer, 2005.

[9] Magnus Persson, Martin Torngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe, and Joachim Denil. A Characterization of Integrated Multi-View Modeling in the Context of Embedded and Cyber-Physical Systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10. IEEE, 2013.

[10] Aditya Shah, Aleksandr Kerzhner, Dirk Schaefer, and Christiaan Paredis. Multi-view Modeling to Support Embedded Systems Engineering in SysML. In *Graph transformations and model-driven engineering*, pages 580–601. Springer, 2010.

[11] Richard Paige, Phillip Brooke, and Jonathan Ostroff. Metamodel-Based Model Conformance and Multi-view Consistency Checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):11, 2007.

[12] Martin Fowler and Matthew Foemmel. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[13] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelsteiin, and Ernst Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):28–63, 2003.

[14] Matthew Hause. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.

[15] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2011(E) Systems and software engineering – Architecture description. Technical report, Dec 2011.

[16] Yanja Dajsuren, Christine Gerpheide, Alexander Serebrenik, Anton Wijs, Bogdan Vasilescu, and Mark van den Brand. Formalizing Correspondence Rules for Automotive Architecture Views. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, pages 129–138. ACM, 2014.

[17] Thomas Johnson, Aleksandr Kerzhner, Christiaan JJ Paredis, and Roger Burkhart. Integrating models and simulations of continuous dynamics into sysml. *Journal of Computing and Information Science in Engineering*, 12(1):011002, 2012.

[18] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2010.

[19] Sebastian Herzig, Ahsan Qamar, and Christiaan Paredis. An approach to Identifying Inconsistencies in Model-Based Systems Engineering. *Procedia Computer Science*, 28:354–362, 2014.

[20] Clare Gryce, Anthony Finkelstein, and Christian Nentwich. Lightweight Checking for UML Based Software Development. In *Workshop on Consistency Problems in UML-based Software Development., Dresden, Germany*, 2002.

[21] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[22] Stefan Feldmann, Sebastian Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan Paredis, and Birgit Vogel-Heuser. A Comparison of Inconsistency Management Approaches Using a Mechatronic Manufacturing System Design Case Study. In *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 158–165. IEEE, 2015.

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

Chapter 9

Paper D: Co-evolution of Simulink Models in a Model-Based Product Line

Robbert Jongeling, Antonio Cicchetti, Federico Ciccozzi, Jan Carlson
In the proceedings of the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS 2020)

Abstract

Co-evolution of metamodels and conforming models is a known challenge in model-driven engineering. A variation of co-evolution occurs in model-based software product line engineering, where it is needed to efficiently co-evolve various products together with the single common platform from which they are derived. In this paper, we aim to alleviate manual efforts during this co-evolution process in an industrial setting where Simulink models are partially reused across various products. We propose and implement an approach providing support for the co-evolution of reusable model fragments. A demonstration on a realistic example model shows that our approach yields a correct co-evolution result and is feasible in practice, although practical application challenges remain. Furthermore, we discuss insights from applying the approach within the studied industrial setting.

9.1 Introduction

When Darwin proposed his theory of evolution by natural selection, he famously concluded that “from (so) simple a beginnings endless forms most beautiful and most wonderful have been, and are being *evolved* [1].” In software engineering today, gradual and parallel changes are applied to software models with the goal of spawning variants addressing diverse requirements. The individual evolution and collective co-evolution of these variants need to be managed to ensure continued opportunities for reuse. To this end, software product line engineering (SPLE) proposes to organize development artifacts and their variants in product lines [2]. In this work, we study an industrial setting with a model-based product line where Simulink models are used to design and implement software components for the development of complex embedded systems.

Simulink is one of the most-used tools for model-based development of embedded systems [3]. It is a MATLAB-based graphical modeling environment with extensive support for simulations and code generation. Simulink models are created by defining a data flow by linking predefined and custom blocks. These blocks can represent defined functions such as logical operators or arithmetic operations, but also more complex functions such as integrators or look-up tables. A special type of block, called subsystem, can, in turn, contain a set of connected blocks, thereby providing the possibility of hierarchically organizing models.

In the studied industrial development setting, reuse of (parts of) models for software components is promoted to reduce the lead time for their development and maintenance. A known best practice for managing variants in a product line is through feature models, which shows the different variants present in the product line and the points at which the product can vary. Nevertheless, a commonly observed approach in industrial settings is to skip the creation of a feature model and instead start by copying assets from an existing project to reuse them in another project, leading to so-called *clone-and-own* product lines [4]. This kind of reuse is commonly used in industry because it requires no initial investment and it is simple to start with. Its downside is the lack of systematic reuse, which is required to fully benefit from the advantages of product lines [5].

In the SPLE paradigm, common functionality is contained in a *platform* from which various related products can subsequently be derived. Different products can thus be branched off from the platform and further developed to fulfill their unique requirements. This setup allows for customization of individual products while benefiting from organized reuse of common functionalities, which can centrally evolve. Indeed, a platform is expected to be periodically revised, for example, to fix bugs, or to include software for new or changed requirements. Such an evolution in the platform may need to be propagated to the derived products to keep them consistent with the platform. In other words, the derived products may need to co-evolve.

In typical model-driven engineering (MDE) scenarios, co-evolution refers to the need to update models upon a change to the metamodels they conform to. Automation of co-evolution of metamodels and models may utilize these conformance relationships [6]. This is one of three relationships typically considered in co-evolution. The other two are (1) a relation between a model transformation and a metamodel, and (2) an indirect dependence of a model on a metamodel [7]. In this paper we consider the need for derived products to co-evolve, upon an evolution of the platform, to maintain the integrity of the product line. In the studied setting, derived products and platform are all Simulink models.

The setting differs from metamodel-model co-evolution due to the following three reasons:

1. the relation between the models is not well-defined (at some point in the past, the derived product was branched off from the platform and after that has possibly undergone separate revisions, as illustrated in Figure 9.1);

2. there is no traceability of reuse, so it is not known which portions of which artifacts should co-evolve; and

3. derived products are not necessarily co-evolved, the changes in the platform might not be adopted depending on the requirements for that specific derived product.

In the studied industrial setting, analyses to find re-used model portions and assessment of the impact of propagating changes between them are currently

performed manually. It is a time-intensive, difficult, and error-prone task. The choice on whether to propagate co-evolution changes to a product is based on the nature of the change, how the product differs from the platform, and on the specific requirements of the product. Some of the engineers working on the Simulink models are in charge of making this decision, we refer to them as domain experts. We aim to provide automated support for domain experts, at the granularity of reused model fragments, i.e. a group of connected model elements. This paper contributes an approach and insights from its application on real models. Furthermore, some of the proposed techniques may contribute to moving from clone-and-own approaches to product lines

The remainder of this paper studies the evolution of the platform and how to co-evolve the derived products. Section 9.2 describes the context of the work in terms of the studied industrial setting and describes the studied problem. Our proposed approach is outlined in Section 9.3, its implementation and a feasibility study in the form of its application on a realistic example are presented in Section 9.4. Insights from this process as well as the application of the approach in an industrial setting are discussed in Section 9.5. Related research works are listed in Section 9.6 and the paper is concluded in Section 9.7.

9.2 Motivation

9.2.1 A clone-and-own Software Product Line

The studied setting is a development team at our industrial partner, responsible for the design, implementation, and testing of control software for a set of embedded systems. To enhance the opportunities for reuse of software components and their test cases, the team has adopted SPLE. Their product line is organized as one *platform*, which contains common functionality for products, and several derived products that are branched off from the platform throughout the revision history and then further developed, as illustrated in Figure 9.1. When a new requirement comes in, domain experts decide whether it is cross-product, and therefore to be implemented in the platform, or if it is specific to one of the products, and therefore to be implemented in that specific product only. New revisions of the platform are released periodically. Upon such a release, the

new or changed functionality in the platform may need to be propagated to the derived products.

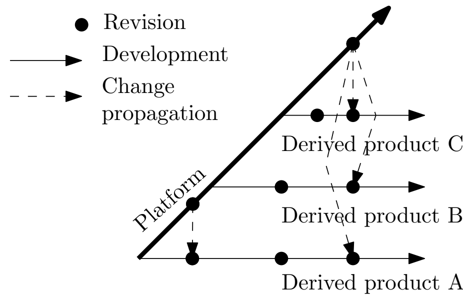


Figure 9.1. Organization of product line in a platform and derived products (A, B, C). Upon a new release of the platform, changes made in the platform since the last revision may need to be propagated derived products.

Currently, design and implementation of control software are done by the same team, using model-based development. The developed products are comprised of software components that are implemented in *Simulink* models. Each of these models is associated with a test harness and test cases. The eventually deployed C code is automatically generated from these models using *Embedded Coder*. Real-time behavior of the generated code is studied in hardware-in-the-loop tests, which are later followed by tests on lab hardware, and eventually on the real deployment target. In this use case, we specifically focus on the development of software components and their test cases.

It is worth noting that not all typical SPLE practices have been adopted in the studied setting, for example, there is no feature model, nor does the development make use of overloaded (150%) models. Overloaded models are models that contain a combination of all possible alternatives and from which a particular variant can be obtained by stripping away the irrelevant elements. Instead, the studied setting can be characterized as a clone-and-own product line, in which the platform contains re-usable components that are copied to the derived products. Typically, a component in a derived product is a reduced version of that component from the platform, or it is cloned, i.e., a one-to-one copy. However, in addition to the reduced and cloned components, derived

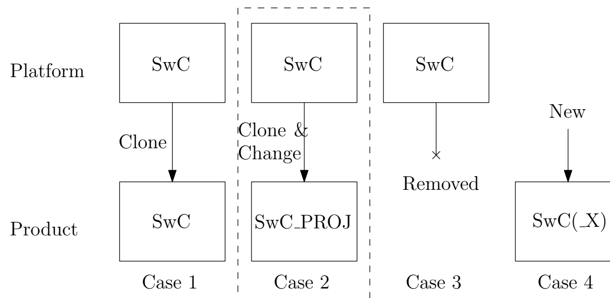


Figure 9.2. Encountered cases of relationships between software components in the derived products and their counterparts in the platform.

products can also contain "new" components that do not exist in the platform and are only relevant for that specific project. Furthermore, derived products may contain modifications of platform functionality. Hence, in the studied setting, the platform is something *almost, but not quite, entirely unlike* a 150% model.

Figure 9.2 shows the four different relationships encountered between software components in the platform and the derived products. In this work, we are particularly interested in case 2, in which components are copied from the platform and then edited for specific use in a derived product. This case is primarily interesting from the co-evolution perspective since common and product-specific functionalities can both be present in software components. Changes to components in the platform might need to be propagated to the derived products. But it is no longer clear how those changes should be propagated because the common portions between platform and products are no longer identical nor are components in the products trimmed-down versions of those in the platform. As mentioned earlier, the absence of a well-defined relationship between the models complicates the co-evolution process, since it cannot rely on e.g. a conformance relation as in the co-evolution of metamodels and models.

9.2.2 Making Software Changes

The systems under development are safety-critical, which means that all developed products require certification according to domain-specific industry

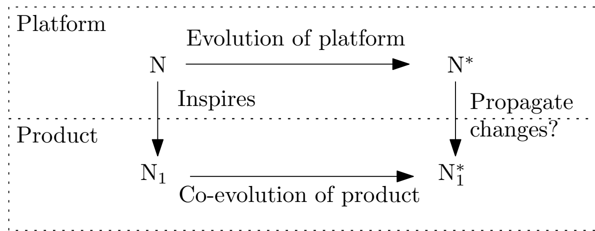


Figure 9.3. Schematic overview of the relation between platform evolution and product co-evolution.

safety standards as well as local and national regulations. Safety assessments are time-intensive and costly. Consequently, a large development effort lies in testing and certifying the software components. One of the improvements to this process was the introduction of a software product line because certification of products can then be based on existing certifications of the platform. Within the current development setting, upon an update to a software component in the platform, a review is needed for each of the products using that component, to assess if they can and should be updated as well.

A schematic co-evolution scenario is illustrated in Figure 9.3. Let N denote a model of a platform component and N_1 a cloned and subsequently modified copy of N in a derived product. Now consider an evolution of the platform where N changes. The evolved model is denoted as N^*. Since N_1 is based on N and probably contains cloned parts of N that are now updated, we may need to co-evolve the model of the product component too, thereby creating N_1^*.

Assessing the impact of a change in the platform on the products requires knowledge of how products are updated after that change. First, it is checked which products use the updated platform component. Then, for each of those products, an expert assesses the need for updating it (bug fixes are more likely to be propagated to the products than new functionality). Currently, this assessment process is completely manual, but given the scale and growth of the product line, this has become overwhelming, tending to infeasible. The platform contains about 180 software components, each of which is implemented as Simulink models and is associated with its own test harnesses and test cases. The mean number of top-level blocks in such a model is 40, with a standard deviation of

24. In the near future, the team will be working on six products derived from the platform. Given that an experienced engineer can check approximately one component every ten minutes, a new release of the platform causes a workload of one month for these reviews only. Secondly, upon a decision to propagate the changes to the product, the engineer manually updates the software components in the product. After the changes are incorporated, the product needs to be retested. Since changes might include new functionality, retesting may also require the development of new test cases or modifications to existing ones. In some cases, the updated test cases can be taken directly from the platform, but this cannot be guaranteed in a general case, due to the informal relationship between platform and product described earlier. Although the changes are performed manually, most effort lies in fact in reviewing the tests to make sure that they are still relevant and complete after a change to the software component and, in case they are not, to update them.

The overall goal of our research is to provide means by which the review process can be reduced and the number of tests needed when a component is updated can be limited. In our work, we focus on the first part of the review process, deciding whether platform changes should be propagated to products or not. The context of our work is thus the assessment of the impact of changes in the platform on software components and test cases in the derived products.

9.2.3 Co-evolution in the Product Line

One way of considering the co-evolution problem is by seeing it as a three-way-merge. In three-way merging, three revisions of an artifact are merged into one. This is commonly used to version development artifacts that are collaboratively developed, in cases where local changes must be merged into an artifact that has in the meanwhile also been changed by someone else. One of the three revisions is considered the "base" artifact, from which the others are derived. These are usually named "theirs", for the remote revision, and "mine" for the local revision. In our case, we could consider the original platform (N) as the "base" model, the evolved platform (N^*) as "theirs", and the product before co-evolution (N_1) as "mine". A three-way merge is then expected to yield the co-evolved product as the "target" (N_1^*).

Three-way merging is a conceptually valid approach of ending up with

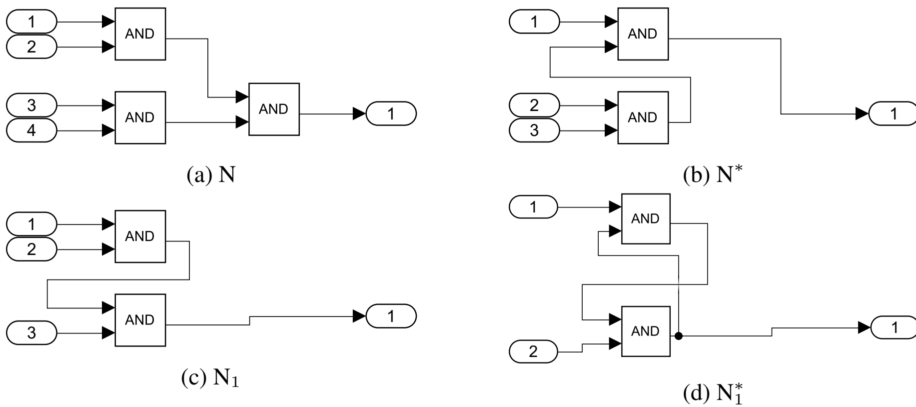


Figure 9.4. Example of a three-way-merge of Simulink models with a semantically incorrect result without raising merge conflicts. The arrangement of the four models corresponds to Figure 9.3.

the co-evolved product [8]. Note however that this merge implies that the target contains all changes as made in the platform. In our scenario, we do not necessarily want to achieve that. Rather, the engineers should be in control of the co-evolution and choose which changes should or should not be propagated to the products. Furthermore, although Simulink contains native support for three-way merging, we found that in some cases using this support results in semantically incorrect or irrelevant target models. An example of this issue is shown in Figure 9.4, where a base model was altered into two semantically identical, but syntactically different, ways. The resulting three-way merge is nonsensical since the input of each of the two *AND* gates depends on the output of the other. Typically the expected output from this type of input would be a merge conflict, which then requires manual resolution, in this example however, the merge is performed without complaints.

Given that we do not always want to propagate all changes, and, in case of a merge, a manual resolution of merge conflicts may be required anyway, we might consider a different approach. We hypothesize that instead of merging models at the level of individual boxes and lines, the level of abstraction can be raised to that of portions of functionality contained in fragments of models, akin

to the idea of feature-based product line evolution [9], but without the creation of a feature model. Within this scheme, an expert can choose to propagate changes to a model fragment encompassing a certain functionality, when this fragment is reused in different products. Assuming this support on the level of portions of functionality, it would be easier to create the co-evolved product, automatically assess the impact of the change, and thus aid in the assessment of the need for propagating it to multiple derived products.

In conclusion, we study the co-evolution of a platform and the products derived from the platform. Our primary focus lies in assisting engineers in assessing change propagation at the level of model fragments rather than low-level blocks, leading to the simplified creation of the evolved derived product (N_1^*).

9.3 Approach

The approach outlined in this section aids domain experts in co-evolving derived products upon evolution in the platform, by accepting or rejecting changes at the level of model fragments. Our approach is based on the observation that the product line is created by a "clone-and-own" approach, i.e., products are derived by copying the platform and then customizing them. Most software components in the products are expected to be a trimmed-down version of the corresponding software components in the platform, although the customization can also entail additions or modifications. Hence, the product software components are typically expected to contain reduced parts of functionality as compared to the platform.

We consider the schematic example models in Figure 9.5. The models on the top row (Figures 9.5a and 9.5b) represent the platform before and after evolution. The models in the first column of the two bottom rows (Figures 9.5c and 9.5e) represent two derived products before co-evolution. Within these figures, the shapes represent model elements. Hence, the example evolution from N to N^* shows an addition, a deletion, and a modification of a model element.

The overall approach consists of the four steps listed in Table 9.1. The last three steps represent a common software engineering pattern of packing, then doing something on the packed thing, and then unpacking again. This section

further details each step using schematic example models. Section 9.4 details the implementation and feasibility study of our approach on a Simulink model that provides a realistic[1] representation of a typical model from the studied development setting.

Table 9.1. High-level overview of the steps in our approach, elaborated throughout Section 9.3 and exemplified throughout Section 9.4.

Step 1	Detect clones between platform and derived products.
Step 2	Replace clones with subsystem references.
Step 3	Evolve the referenced subsystem.
Step 4	For each reference, revert or expand the subsystem.

Step 1 The first step of the approach is to find common functionality shared between product and platform. This step can be skipped when traceability information tracking reuse already exists but is needed in our case since no explicit knowledge of reused functionality is present in the artifacts. In our setting, we expect the origin of common functionality between different files to be through copying. Therefore, we start by looking for exact clones of model portions between platform and products, although differences in layout are acceptable as long as they do not change the semantics. The clones across the three models before evolution are illustrated by dashed lines in Figure 9.5, for N, N_1, and N_2 in Figures 9.5a, 9.5c, and 9.5e respectively. If desired, these results can be stored for future instances of co-evolution, although it should be noted that the derived products may evolve individually in the meantime and in that process break the links that are created in this step.

Step 2 The second step comprises of "packing" each of the cloned model fragments in separate subsystems. Crucially, these newly created subsystems are stored in a common library. In each derived product containing the cloned model fragment, that fragment is replaced by a reference to the created subsystem.

[1]Note that, for obvious reasons related to IP, we could not provide the "real" industrial model in the paper.

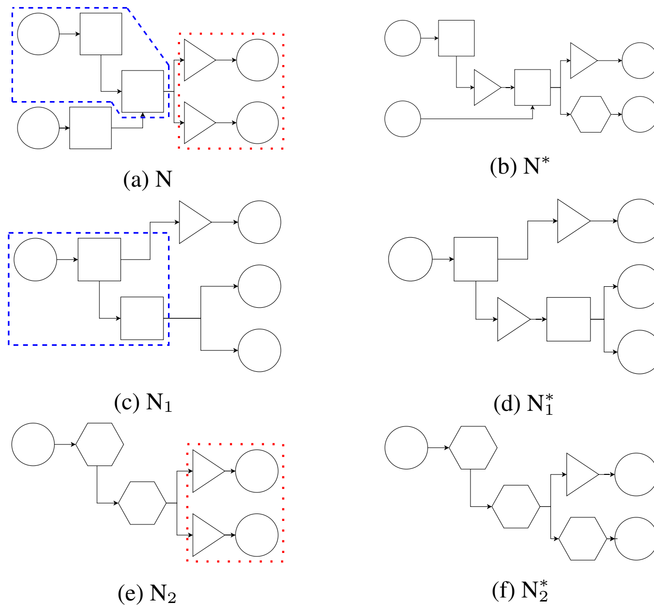


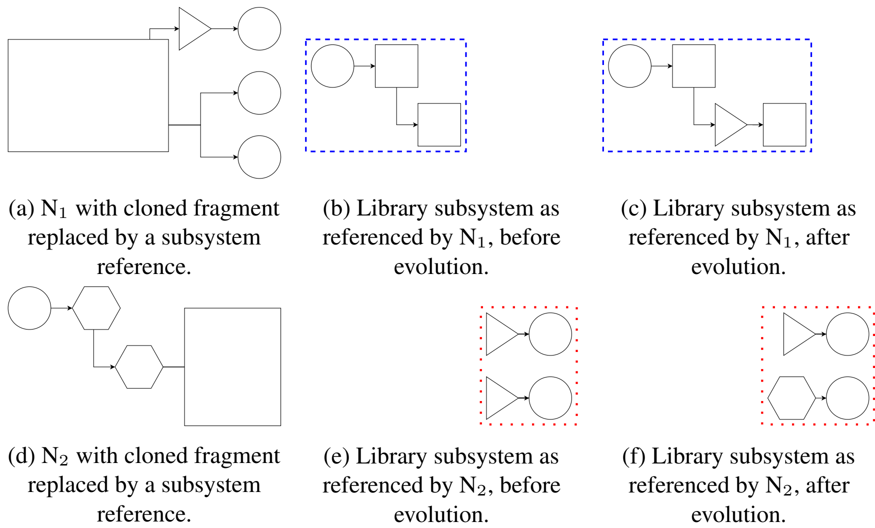
Figure 9.5. Schematic example of models in a platform (N), its evolved version (N^*), and two derived products before (N_1 and N_2) and after (N_1^* and N_2^*) co-evolution. Clones between the platform and derived products are indicated by the dashed and dotted regions in the models before evolution (N, N_1, and N_2).

The results of this step on the products N_1 and N_2 are shown in Figures 9.6a and 9.6d, respectively. Figures 9.6b and 9.6e show the subsystem as created in the common library and referenced by N_1 and N_2, respectively.

Step 3 Now, in all derived products, the cloned fragments have been replaced with a reference to the library subsystem containing that functionality. The third step applies the evolution of the platform to that library subsystem and, by these means, the change is automatically propagated to all derived products. To perform this step, we need to know the following four things:

(a) What evolution happened in the platform;

(b) Which cloned fragments are affected;



(a) N_1 with cloned fragment replaced by a subsystem reference.

(b) Library subsystem as referenced by N_1, before evolution.

(c) Library subsystem as referenced by N_1, after evolution.

(d) N_2 with cloned fragment replaced by a subsystem reference.

(e) Library subsystem as referenced by N_2, before evolution.

(f) Library subsystem as referenced by N_2, after evolution.

Figure 9.6. Cloned functionality is replaced with subsystem references (step 2). Then changes are applied in those referenced subsystems (step 3).

(c) How we can evolve the library subsystems;

(d) For each subsystem reference, whether the change should be accepted or discarded.

The first input we require is to know what evolution happened in the platform between its closest previous release and the current release. This can be obtained using standard model differencing provided by version control systems. Although notoriously difficult for graphical models, Simulink provides effective support for graphical model differencing. Note that the products, between their derivation from the platform and the new platform release, may have evolved on their own too. Therefore, when detecting clones, we consider the latest version of the derived product, rather than the version at the time of branching off from the platform.

Given an evolution of the platform, we need to determine which reused fragments are affected. In the schematic diagrams, we simply draw boxes around cloned fragments (as in Figure 9.5), providing clear clone borders. Naturally, this

kind of meta-information is not present in the output of the clone detection step, nor it can be expected to be part of other traceability mechanisms. Nevertheless, it is important to assign borders to cloned fragments, since we want to limit the applied changes in this step to within cloned fragments, thereby allowing the engineers to adopt the changed fragment as a whole (as in Figure 9.6). Moreover, limiting changes to within fragment borders prevents problems when later (in step 4) "unpacking" the subsystem references again to their constituent blocks. Therefore, we define clone borders by the components within it, i.e., after a change, an added component shall be considered to be within the clone border when it is only connected to components already in the clone. In case of multiple new components, the same definition applies recursively. Modified components are considered within the clone scope if and only if they were in the clone scope before the evolution. Note that this step might result in cases where the change cannot be applied completely since it happens across the boundaries of the cloned fragment. We do not provide support for these specific cases, which are expected to be very few, given the nature of the clones in the studied setting.

Once we know which cloned fragments have changed and how they have changed, we want to apply the corresponding evolution to the library subsystems. For each clone, we can perform the same evolution as it happened in the platform, based on the obtained difference between the current and previous platform releases. Conceptually, this is similar to applying a patch as obtained through a Git *diff*. This is challenging in general for graphical models and not supported in Simulink.

Figures 9.6c and 9.6f show the situation after the platform changes are applied to the library subsystems. Note that in this example each subsystem contains a single change and that one of the changes in the platform, that falls outside the clone boundaries, is not propagated to any subsystem.

After this step, we can "unpack" the subsystems again and obtain the co-evolved models. Before doing so, test cases and calculations of other model metrics can be executed; this can help the engineer to decide if this change shall be propagated to the product, or if it shall be discarded and thereby revert to the original subsystem. This assessment, labeled as step 3d, requires domain knowledge and is therefore always manual.

Step 4 In step four, we break the references to the common subsystem, making the subsystems, in each model instance, unique. Furthermore, we replace the subsystems with their content, thus restoring the models to the same structure as before the co-evolution (but with different contents). Instead of these unpacking actions, we might consider keeping the now uncovered traceability links between clones. We opt to unpack the subsystems back to their constituent blocks to keep allowing the derived products to evolve independently of the platform too. Figure 9.5 shows the evolved products N_1^* (Figure 9.5d) and N_2^* (Figure 9.5f). Note that, as expected, changes outside the cloned fragments are not propagated to the products.

9.4 Feasibility study

In the first part of this section, we describe the implementation[2] of each of the steps as outlined in Section 9.3. Individual steps are supported by fully automated means, except for step 3, which requires an engineer's decision. The different steps are implemented as prototypes, we have not yet created automation to combine the steps into a single executable script. Since the models at our industrial partner cannot be shared, the second part of this section describes the application of the implemented approach on a realistic (not real) model. The third subsection describes experiences from applying the approach to real industrial models.

9.4.1 Implementation

Step 1 In the studied setting, no traceability information indicating reuse of model fragments exists. However, due to the "clone-and-own" nature of the product line, we expected to find exact clones across models. To detect them, we considered three alternatives for clone detection. The first is a built-in Simulink feature that can detect similar subsystems within the same model. In our case, we found that the models are quite "flat", i.e., they typically do not contain

[2]All mentioned scripts and models are available in the following GitHub repository: https://github.com/RobbertJongeling/Simulink-PL-co-evo

subsystems. Furthermore, we aim to find clones across different models, not within one single model.

After that, we considered the SIMONE Simulink code cloning tool [10]. SIMONE can be configured to look for similar subsystems or similar models. Its strength is the ability to detect near-miss clones. In this use case, we expect exact clones at block-level, which is why we eventually opted for the *ConQAT* [11] tool. ConQAT is used in several research works that consider Simulink clones, e.g., to detect anti-patterns [12], or to compare performances of cloning detection tools [10]. It can find the exact clones of model fragments across models by considering the Simulink models as graphs and matching identical sub-graphs. The tool relies on the textual storage format of Simulink models, so it requires input models to be saved as *.mdl*. For practical use, the main challenge is to configure the correct minimum size of to-be-detected clones in order to detect meaningful fragments. Through experimenting on our set of industrial models, we found out that a minimum of 5 blocks yields reasonable results.

Step 2 Step 1 yields cloned model fragments across several models (the platform and one or more derived products). In step 2, we want to convert these fragments into references to a common library subsystem. To do that, we first create a subsystem out of the cloned fragment. Executing the command `Simulink.BlockDiagram.createSubsystem(handles)` creates a subsystem from a list of handles of blocks making up a cloned fragment. This list is the result of step 1. To be able to be converted to a subsystem reference, Simulink requires a subsystem to be *atomic*.[3] This can be achieved by setting the parameter `TreatAsAtomicUnit` of a subsystem to `on` . After that, the subsystem can be replaced by a reference by executing the command `Simulink.SubSystem.convertToModelReference('ModelName/SubsystemName', 'libref', 'ReplaceSubsystem', true)`.

Note that after these actions, a single cloned fragment in one of the models is replaced with a subsystem. However, in cases where a fragment is cloned across multiple derived products, an extra step is required to put a subsystem reference in each of them. First, a subsystem reference is created in each of the target

[3]The execution of blocks in the atomic subsystem can not be interleaved with the execution of blocks outside it.

models. In this scenario, we want to make all the subsystem references to point to the same library subsystem. This is achieved by changing the `ModelFile` parameter of subsystems to point to one library file containing the common subsystem.

Step 3 After step 2, all clones are replaced by subsystem references pointing to the same library subsystem. Consequently, any changes in that library subsystem during step 3 are automatically propagated to all models containing a reference to it. Now we need to apply the evolution that happened in the platform and within the clone to the subsystem library.

First, we find out what the change implies. In Simulink, the difference between two models can be obtained using either `visdiff(N,N$^*$)`, which creates a visual comparison and a report, or through `slxmlcomp.compare(N,N $^*$)`, which returns the difference in an `xmlcomp.Edits` object.

However, the latter is also mostly visual, since the object's main purpose is to allow the creation of a comparison report as created by the former command. Nevertheless, the `xmlcomp.Edits` object provides the roots of tree representations of both models and allows a programmatic traversal of them. This makes it possible to automatically look for all changed (parameter `Edited` set to `true`) or new blocks. The object will contain only those nodes that have changed, so all common ones that are unchanged are not mentioned. Using the same traversing technique, we can determine which clones are affected by the particular evolution in the platform and which changes are made to them. This is how the calculation of differences and their localization in code fragments could be implemented in the Simulink environment. However, Simulink does not provide features similar to Git, in which differences between files can be stored into a patch that can later be applied to the original file to obtain the changed one. Therefore, applying the changes is currently a manual process and consequently, there is not any notable added value in automating the remaining sub-steps. Upon the decision to accept the change, we continue with the final step, alternatively, the changes are discarded and the model is returned to the state before the changes made in step 2.

Step 4 Step 4 should break the links to the referenced subsystem and unpack the subsystems back to individual model components. We adopted a script from the Mathworks forums to break the link to the library subsystem and make the subsystem in the derived model unique [13]. After that, we converted subsystems back to their original components through `Simulink.BlockDiagram.expandSubsystem` with the subsystem that should be unpacked as an argument.

9.4.2 Demonstration

We illustrate the implementation by applying it to a public Simulink example model. In selecting the example, we considered models that are as realistic as possible in the sense that they 1) consider pieces of control software that could be used to generate code, 2) have a size and complexity comparable to typical models encountered in the studied industrial setting, and 3) can have derived products and can be subject to additions, changes, and deletions of model elements. We consider as a model the `airflow_calc` subsystem within the `fuel_rate_control` subsystem within the `sldemo_fuelsys` example model [14]. We slightly modified that by changing the condition for enabling the switch from (O2_normal ∧ fuel_mode = LOW) to ((O2_normal ∧ fuel_mode = LOW) ∨ (O2_normal ∧ fuel_mode = HIGH)). The resulting base model (N) is shown in Figure 9.7.

To show a reasonable example of evolution, we made sure to consider different types of changes, both within and outside a cloned fragment. The evolution of the model contains three changes compared to the base.

1. It fixes a bug by adding a negation in the switch condition, thus making it ((O2_normal ∧ fuel_mode = LOW) ∨ (¬ O2_normal ∧ fuel_mode = HIGH)).

2. It refactors two multiplication blocks with 2 inputs each to a single multiplication block with 3 inputs, in the “Feedforward Control” part.

3. It updates the constant value Oxygen Sensor Switching Threshold from 0.5 to 0.7.

The resulting evolved model (N*) is shown in Figure 9.8.

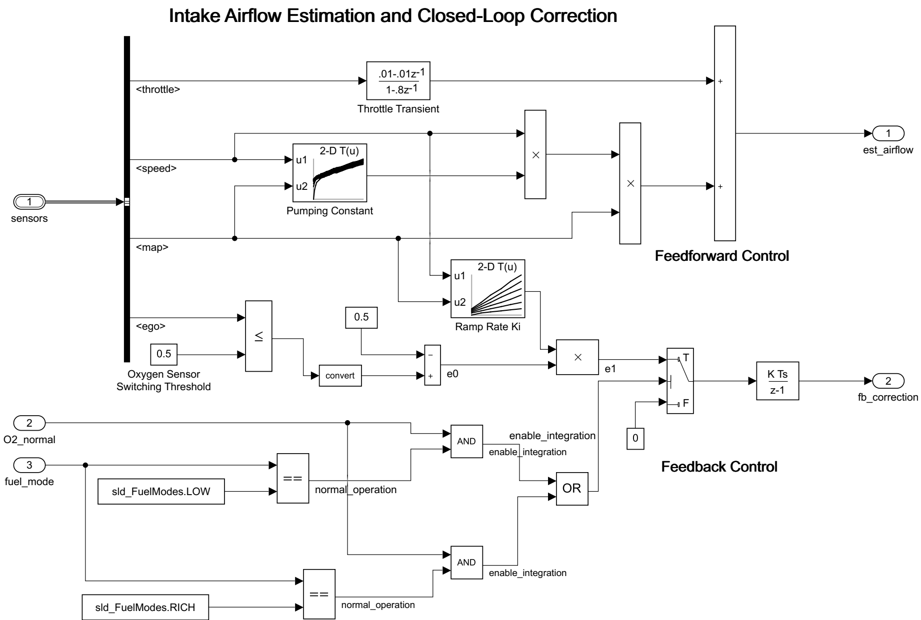


Figure 9.7. Realistic Simulink model as adapted from one of the Simulink examples. This model is considered the base platform model (N) in this Section.

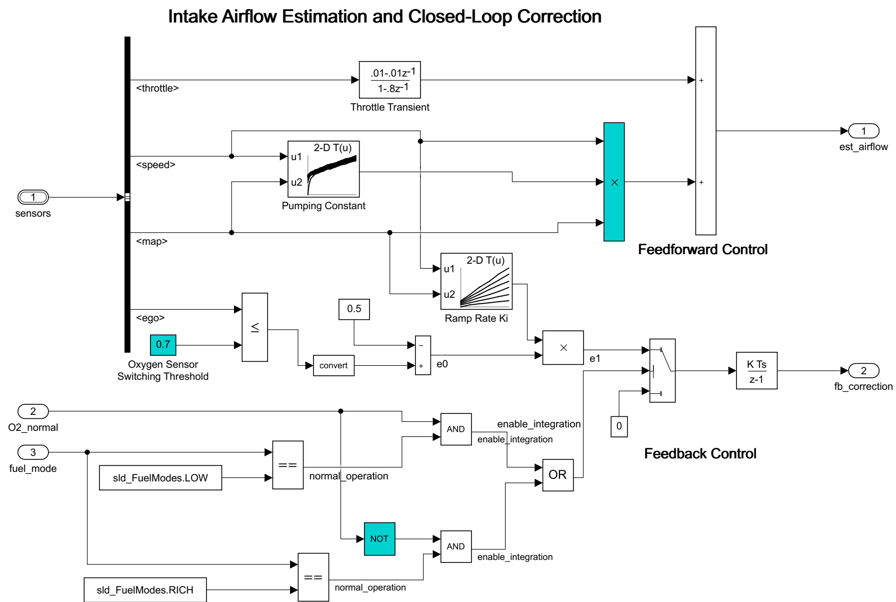


Figure 9.8. Evolved version of N from Figure 9.7, locations of changes indicated in cyan. This model is the evolved platform (N*) in this section.

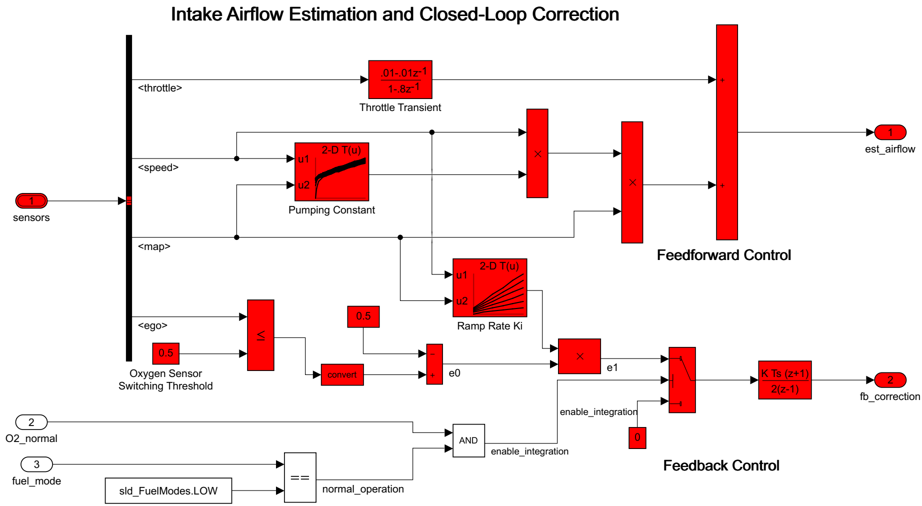


Figure 9.9. Model version derived from N from Figure 9.7. This model is the derived product (N_1) in this section. Highlighted in red are the blocks that form the fragment of N_1 that is cloned from N.

We consider one product derivation from the original platform model. The derived product model is, as typical in our setting, cloned from the platform model and then modified. Two changes were made after the cloning step:

1. Part of the switch condition functionality was stripped, leaving only the original from the `sldemo_fuelsys` example: (O2_normal $\wedge$ fuel_mode = LOW).

2. The integrator method used by the discrete integrator block has been changed from `Forward Euler` to `Trapezoidal`.

The resulting derived model (N_1) is shown in Figure 9.9.

Now we show the steps of our approach and how they could be applied to this use case.

Step 1. We ran one of the ConQAT pre-defined configurations that detect clones between Simulink models: simulink-analysis.cqr. As input, we used *.mdl*

versions of the original model (N, Figure 9.7) and the variant (N_1, Figure 9.9), and we configured a minimum-clone-size of 5. In this example, a single large cloned fragment is identified between N and N_1, it is shown in red in Figure 9.9. Note that the integration method is a property of the discrete integrator block and since the clone tooling works on the level of blocks, the two integrator blocks are marked as clones.

As input for step 2, we require a list of handles of all blocks in the cloned fragment. ConQAT reports a list of block identifiers in the form of a Matlab script (*.m* file) that can be run to obtain the colorization as shown in Figure 9.9. It does this by setting, for each block in the model, the parameter BackgroundColor. To obtain a list of handles, we modified this output file to, instead of setting these parameters, getting the 'handle' parameter for all blocks that are part of the cloned fragment (which they are when their background color is set to anything else than white). So we take the generated simulink-analysis_matlab-colorm-file-writer_Clone_0.m file (and possibly the other files, if more clones were found), and then we run the script to get the block handles out.

Step 2. Now we have all the handles of the blocks in the subsystem. At this point we make a subsystem out of all the cloned blocks, we make it atomic and finally we convert it into a reference:

1. `Simulink.BlockDiagram.createSubsystem(handles);`
2. `set_param('intakeAirflow_var/Subsystem', 'TreatAsAtomicUnit', 'on');`
3. `Simulink.SubSystem.convertToModelReference( 'intakeAirflow_var/Subsystem', 'modelref_lib', 'ReplaceSubsystem', true)`

After applying these steps, model N_1 contains a reference to the library subsystem as stored in `modelref_lib`. The resulting model is shown in Figure 9.10

Step 3. The engineer shall now apply to the library subsystem the changes that are limited to the clone scope, as defined in Section 9.3. Following our rules, the new negation block is not part of the change to be applied, but the changed

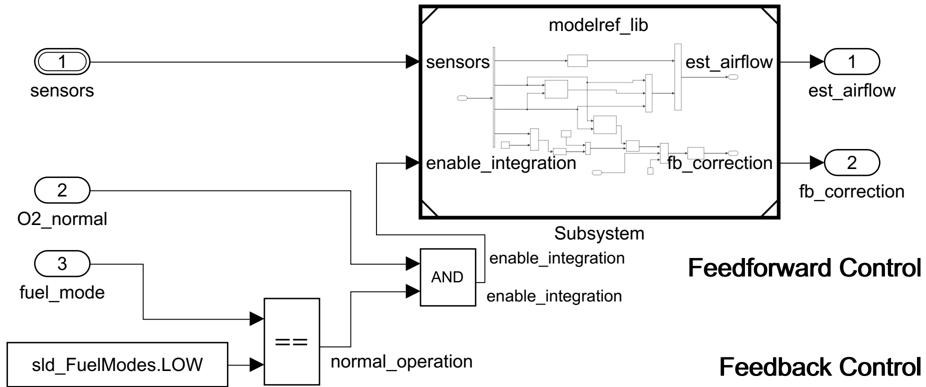


Figure 9.10. N_1 being prepared for co-evolution, clone is replaced by subsystem reference.

parameter and the multiplication refactoring are. The changes made in the library are immediately visible also in the variant since it contains a reference to it. Now that the changes appear in the variant, it is up to the domain expert to choose whether to accept or discard the changes made by the co-evolution mechanism. In this example, we assume that the changes are accepted, to continue with step 4.

Step 4. For breaking the link to the library subsystem, we run the aforementioned script: `Replace_MdlRef_SubSys('intakeAirflow_var')`. As a small note on an implementation detail, this script copies the model and breaks the link in the copy, but this is not essential to the functionality and not relevant for our approach. Now that the link is broken, we can unpack the subsystem again to its constituent blocks through `Simulink.BlockDiagram.expandSubsystem('intakeAirflow_var_new')`. We obtain the model as shown in Figure 9.11, which represents N_1^*, the co-evolved derived product.

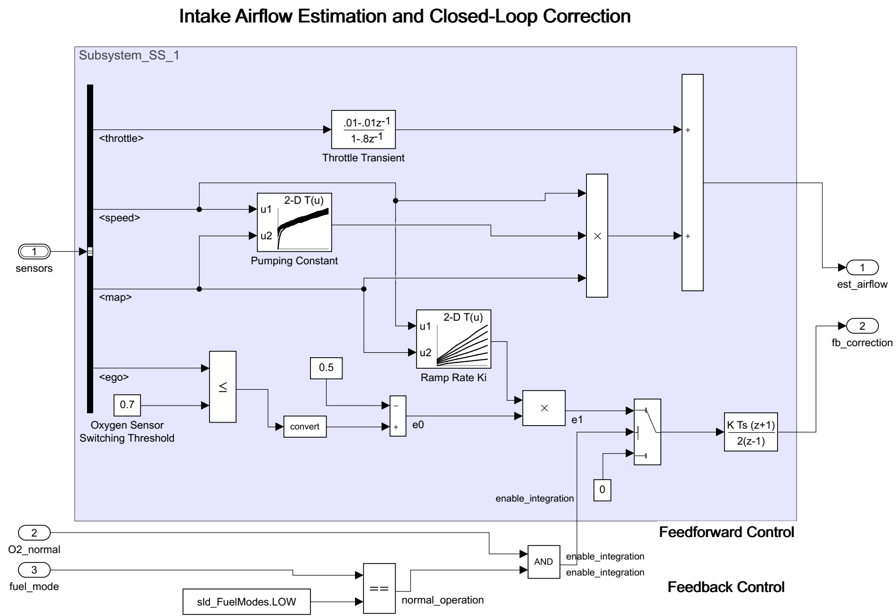


Figure 9.11. The result of the co-evolution approach: N_1^*. Note that the background is there just to clarify the image. It can be removed by setting the additional option 'CreateArea' to 'Off' (available from 2019a)

9.4.3 Experiences

We now describe the outcomes of applying our approach to real industrial models. We discuss in particular two co-evolution cases of software components (referred to as case A and case B) that are particularly interesting because they represent extreme cases in the described setting. The derived products are large reductions of the platform (From 80 to 20 in case A and from 74 to 15 top-level blocks in case B) and contain some changes.

Step 1 In case A, two clones were found, one consisting of eight blocks and another consisting of seven blocks. The platform contains two occurrences of a pattern of a few blocks; one occurrence is included in the clone. It is not clear why one is chosen over the other. Case B contains only a single clone, by the size of ten blocks.

In general, detecting clones using ConQAT worked fairly well for our use case, because we expect identical clones between platform and components. Nevertheless, it is still challenging to properly configure the tool (e.g., the minimum size of clones) to obtain desired results. Another potential hiccup is the requirement of the clone detection tool to provide input models in the textual `.mdl` format, which is not supported for all models, for example, those including internal test harnesses. We did not encounter this problem in our setting since all test harnesses had previously been exported.

An underlying assumption of our approach is that the software component models can be divided into meaningful fragments, representing small chunks of functionality, such as input handling, or a particular calculation. When applying the approach on a set of industrial models, we found that this assumption does not always hold. In particular, smaller models proved less suitable for our approach since the cloned fragments were either too small to be meaningful or not existing at all. Furthermore, for small models, a manual approach could be a better choice time-wise.

Step 2 In case A, since there are two clones, the step to create a subsystem from a cloned fragment has to be executed twice. This works fine, but one thing that could form a barrier for practical use is that the intermediate result can look

messy since the creation of subsystems will cause other lines to rearrange and possibly get entangled.

Step 3 A considerable amount of changes in the revision history are non-functional. They are limited to e.g. updates to parameter names or comments. In case A, the change in the platform is a correction to the model for which an extra multiplication with a constant factor is added to a flow. In case B, also functionality is added to the model, but in contrast to case A, it is partially outside of the cloned fragment and is therefore not propagated by the approach, as described in Section 9.3.

Step 4 In case A, the unpacking functions correctly. It should be mentioned that for practical applications a hinder may be that the layout of the model after applying these steps may differ significantly from the layout before the steps. This holds even if the co-evolution is limited to the updates of parameter values.

Moreover, we have encountered a type of case in which our approach does not help the engineers much. One example concerns a refactoring impacting the entire model, rather than being localized to a specific model fragment. Examples of such refactoring efforts include: starting to monitor values on all output ports for testing purposes, externalizing a model's test harness, or upgrading the model to work with a new Simulink version. In these cases, our proposed approach would likely miss some occurrences, since they would occur outside cloned fragments.

9.5 Discussion

We have shown our implementation of automatic support for co-evolution between models in an industrial setting of a software product line. The implementation can be integrated into the development process since it builds on the same Matlab platform as the Simulink models. Nevertheless, some points in our approach require further research, as identified throughout Section 9.4. For example, a possible scenario is that a single evolution affects several cloned fragments. In that case, propagating a few of them may not yield a meaningful

model, for instance, because an extra value was introduced in one cloned fragment but not used in any other. This is somewhat related to the issue of changes across clone borders that cannot be propagated. In the end, there may remain cases for which the approach does not provide the desired result. However, thus far, these concerns have not impacted the application of the approach, because they did not occur in the studied industrial models. Overall, it is clear that the solution is not the holy grail and that several problems remain open.

In Section 9.4, we showed that the approach still requires human intervention for a few decision-making actions. Ideally, all steps not strictly requiring an engineer's decision would be automated, but the following implementation challenges have prevented us from doing that so far. To automate the application of changes in the library subsystem, it is required to be able to refer to the same block across different models. Handles are not carried over between models so it is required to rely on identifiers. However, most blocks within the models have no explicit name, but they rather get automatically assigned names (e.g. `LogicalOperator4`). This numbering system may completely change in the case of a derived product in which some modifications were made, hence identifiers are also not reliable to identify blocks across models. Even if this would be solved, there is still one action left to be done manually, which cannot be automated. More specifically, the engineer must decide whether to accept or discard the proposed co-evolution. By using our approach, the engineer gets access to more information to base that decision on, since the proposed co-evolution model can be tested and other model metrics derived from it.

When using the approach, an unexpected benefit was to get support for localizing reused fragments. There was no traceability for this in place except for an inconsistently followed naming convention at the level of model files that indicates if they are derived and modified after being branched off from the platform. Although requirements traceability is in place in the industrial setting, it specifically links requirements to Simulink models and test harnesses. The application of the clone detection tooling allowed instead for identification of reuse at sub-model level. In the studied setting, we encountered flat models, in contrast to the common practice in Simulink of creating hierarchies of nested subsystems. The approach is not dependent on the absence of subsystems, but the practical implementation is guided by it. Indeed, the typical absence

of hierarchy in the studied setting and the expectation of finding exact model fragment clones have guided our choice of clone detection tool.

The studied problem exists in part because of the clone-and-own practice and the accompanying lack of traceability between re-used model fragments. As a way forward towards a product line, we mention two main directions. In the first, Step 4 of the approach could be omitted, which would allow the practice to migrate to a more systematic product line. In such a case, the identified clones might be lifted to a variant description model in pure::variants [15], which could then be used to generate different model variants. The second direction would be to have more support for product lining in the modelling language, such that clone-and-own may be avoided. Variability in Simulink can be managed through feature modelling, negative variability, or delta modelling [16]. In our setting, we would want to manage variability outside single models, given the strict regulations on their certification. Therefore, feature modelling and managing variability using pure::variants could be a good alternative in the studied setting, despite the required changes throughout the engineering process to fully benefit from this way of organizing reuse.

It is clear that the proposed approach is heavily guided by the used modelling tooling and practices in the studied setting. Nevertheless, we can imagine generalization to other modelling languages. At a very high level, the approach is not uniquely applicable to Simulink only, we can imagine finding clones between models and then replacing them with references in other languages too. However, this depends also on the storage format of models. For modelling tools with repository-based instead of file-based storage, variability may be addressed in a completely different way, making our approach less applicable. We finalize this discussion by reiterating that, in this work, we address an existing clone-and-own practice and aim to provide means to improve it, despite knowing that it is not the ideal product line practice.

9.6 Related work

Within MDE, co-evolution commonly refers to model-metamodel co-evolution [6]. In contrast, co-evolution in software product line engineering commonly refers to the parallel evolution of a feature model and software model. For example, re-

cent work on co-evolution in model-based product lines enables the co-evolution of a feature model and model variants, through an approach combining the management of revisions of models through time and across variants [17]. In this work, we have considered a third option, the co-evolution of software models belonging to a platform and derived products in a model-based software product line. A key difference between existing co-evolution work and our work is that we cannot rely on the formal conformance relation between artifacts, as is typically done in operator or inference approaches to co-evolution in MDE [18]. Instead, in our case, the derived products are merely inspired by the platform, but there are no more guarantees about the relation between them.

We study the co-evolution of Simulink models between which fragments have been reused through exact copies. Alternatively, Rumpe et al. propose to assess reuse opportunities of Simulink model fragments by checking that the behaviors of different fragments are the same [19]. In their work, they also consider a software product line setting. They note, similar to us, that behaviorally identical fragments across models may be replaced with references to a single library fragment. Due to the expectation of encountering exact syntactic clones in the studied setting, we do not further study the identical behaviors of different models. Other work also considers co-evolution of model fragments [9], but in contrast to our work, they construct a feature model and consider the automatic creation of variants from it. In our case, the variants have already been created and must be co-evolved with the platform from which they are derived. A similar study checks the integrity of the co-evolution by utilizing notations from evolutionary biology that keep track of the variations between the platform and derived products [20]. Other work considers co-evolution of models and libraries [21]. Although that work seems closer to our studied setting, still it relies on a formal conformance relationship between the models and the libraries.

There have been other works considering variability in Simulink models. Dajsuren has proposed to manage clones and their variants outside of models, through configuring clones using a textual language [22, Chapter 7]. The approach allows for reuse of subsystems across different models while keeping the clone management separate from the main model, thus preventing overloaded models from growing too large and becoming unreadable. Another proposal

to prevent overly overloaded models is to define model variants using the set of operations that are required to obtain the variant from the base model, so-called delta-modeling [23]. In this work, we have a different starting point, since we already encounter an existing clone-and-own product line. Neither of the aforementioned approaches specifically considers co-evolution of the cloned fragments. Other work has specifically explored three-way-merging as a solution to co-evolution in software product lines [8]. As we have argued, there is an inevitable need for human intervention in those cases and as we then hypothesize, resolving conflicts would be easier when the engineer can reason about them at the level of functionalities rather than individual blocks.

There has also been some work towards impact analysis [24] and test evolution [25] for Simulink models in an industrial setting. Since the proposed approaches and implementations in those works are tightly integrated with the industrial setting in which they are developed, their results are not so easily generalized. The impact analysis work assesses the impact of changes in a model on their tests by forward and backward traversal of the flow in the model, until reaching input and output ports. All tests involving those inputs or outputs are then said to be impacted. The test evolution work builds further on that test identification work to generate test harnesses in case the evolution requires the creation of a new one. Hence, the work limits consideration of co-evolution to tests. This is one of the aspects we aim to consider in our future work, whereas in this paper we have focused on co-evolution of software models.

Related work on clone management [26] emphasizes its need but focuses on code-based software representations. Within code clone analysis, evolution of code clones is often studied with the aim of visualizing it [27]. Moreover, those studies have reported disagreeing findings from empirical studies [28]. In addition to these code-based clone evolution studies, other works have considered graph-based (Simulink) models as well. ModelCD is a clone detection tool based on ConQAT that can detect exact and near-miss clones across Simulink models [29]. The evolution of cloned Simulink fragments has been studied before too [30]. In that work, the authors focus on identifying clones across revisions of Simulink models to study their evolution.

9.7 Conclusion

We have presented our approach for the co-evolution of Simulink models in a model-based product line. Co-evolution is considered at the level of model fragments that are cloned across a platform and derived products. In this work, we aimed to provide support for the process of assessing whether or not a change should be propagated. The feasibility of our approach is shown on a realistic example model. We showed that our approach yields correct results and that automated support aids in the decision-making process.

To improve the practical applicability of this work, we plan to extend our approach to include “what-if” analysis on the generated co-evolved products. This kind of analysis aids engineers in assessing whether to accept or discard a change and ascertaining how associated test cases should be updated given a change to a model. We will then build upon that for another planned future work, where we plan to provide more support for semi-automatically co-evolving the test cases as well. This will contribute to one of our main goals, i.e. reducing the effort of review and test upon platform evolution. Another interesting future direction is to consider alternatives for the code clone detection step. For that, we are looking into combining our work with other research efforts carried out at our partner company which focus on the automatic identification of potential model fragment reuse based on requirements similarity.

Bibliography

[1] Charles Darwin. *The origin of species*. PF Collier & son New York, 1909.

[2] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[3] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[4] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110, 2013.

[5] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400. IEEE, 2014.

[6] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231. IEEE, 2008.

[7] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. What is needed for managing co-evolution in MDE? In *Proceedings of the 2nd*

International Workshop on Model Comparison in Practice, pages 30–38, 2011.

[8] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. Aligning coevolving artifacts between software product lines and products. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 9–16, 2016.

[9] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 85(10):2261–2274, 2012.

[10] Manar H Alalfi, James R Cordy, Thomas R Dean, Matthew Stephan, and Andrew Stevenson. Models are code too: Near-miss clone detection for simulink models. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 295–304. IEEE, 2012.

[11] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 603–612. IEEE, 2008.

[12] Matthew Stephan and James R Cordy. Identification of simulink model antipattern instances using model clone detection. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 276–285. IEEE, 2015.

[13] MathWorks Support Team. How can i replace my model reference block with a subsystem in simulink? https://www.mathworks.com/matlabcentral/answers/98940-how-can-i-replace-my-model-reference-block-with-a-subsystem-in-simulink, oct 2013.

[14] MathWorks. Modeling a fault-tolerant fuel control system. https://mathworks.com/help/simulink/slref/modeling-a-fault-tolerant-fuel-control-system.html, 2017.

[15] Danilo Beuche. Variant management with pure:: variants. Technical report, Technical report, pure-systems GmbH, 2003. http://www. pure-systems. com, 2003.

[16] Aitor Arrieta, Goiuria Sagardui, and Leire Etxeberria. A comparative on variability modelling and management approach in simulink for embedded systems. *V Jornadas de Computación Empotrada, ser. JCE*, (26-33), 2014.

[17] Felix Schwägerl and Bernhard Westfechtel. Integrated revision and variation control for evolving model-driven software product lines. *Software and Systems Modeling*, 18(6):3373–3420, 2019.

[18] Richard F Paige, Nicholas Matragkas, and Louis M Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016.

[19] Bernhard Rumpe, Christoph Schulze, Michael Von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral compatibility of simulink models for product line maintenance and evolution. In *Proceedings of the 19th International Conference on Software Product Line*, pages 141–150, 2015.

[20] Anissa Benlarabi, Bouchra El Asri, and Amal Khtira. A co-evolution model for software product lines: An approach based on evolutionary trees. In *2014 Second World Conference on Complex Systems (WCCS)*, pages 140–145. IEEE, 2014.

[21] Luca Berardinelli, Stefan Biffl, Emanuel Maetzler, Tanja Mayerhofer, and Manuel Wimmer. Model-based co-evolution of production systems and their libraries with automationml. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8. IEEE, 2015.

[22] Yanjindulam Dajsuren. *On the design of an architecture framework and quality evaluation for automotive software systems*. PhD thesis, 2015.

[23] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-class variability modeling in

matlab/simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 1–8, 2013.

[24] Eric J Rapos and James R Cordy. Simpact: Impact analysis for simulink models. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 489–493. IEEE, 2017.

[25] Eric J Rapos and James R Cordy. Simevo: A toolset for simulink test evolution & maintenance. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 410–415. IEEE, 2018.

[26] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Clone management for evolving software. *IEEE transactions on software engineering*, 38(5):1008–1026, 2011.

[27] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[28] Jeremy R Pate, Robert Tairas, and Nicholas A Kraft. Clone evolution: a systematic review. *Journal of software: Evolution and Process*, 25(3):261–283, 2013.

[29] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Complete and accurate clone detection in graph-based models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 276–286. IEEE, 2009.

[30] Matthew Stephan, Manar H Alalfi, James R Cordy, and Andrew Stevenson. Evolution of model clones in simulink. In *ME@ MoDELS*, pages 40–49, 2013.