Software and Systems Modeling
https://doi.org/10.1007/s10270-022-01000-5

THEME SECTION PAPER

Consistency management in industrial continuous model-based development settings: a reality check

Robbert Jongeling[1] · **Federico Ciccozzi**[1] · **Jan Carlson**[1] · **Antonio Cicchetti**[1]

Received: 31 October 2021 / Revised: 27 January 2022 / Accepted: 14 March 2022

Abstract
This article presents the state of practice of consistency management in thirteen industrial model-based development settings. Our analysis shows a tight coupling between adopting shorter development cycles and increasingly pressing consistency management challenges. We find that practitioners desire to adopt shorter development cycles, but immature modeling practices slow them down. We describe the different patterns that emerge from the various industrial settings. There is an opportunity for researchers to provide practitioners with a migration path toward practices that enable more automated consistency management, and ultimately, continuous model-based development.

Keywords Model-based development · Consistency management · Agile

1 Introduction

We define continuous model-based development (MBD) as developing in short cycles using models as core development artifacts for system design, software implementation, or both. Continuous integration [36] and Agile [8] practices have gained popularity in conventional software development. We hypothesize that developing in shorter cycles and frequently integrating work is beneficial in MBD too. A key difference between our scope and the Agile modeling paradigm [3] is that we consider models as core development artifacts, rather than just aids to the agile development process.

In our scope, we consider MBD to entail those practices in which models are core development artifacts during system and software development, i.e., models are developed and maintained throughout the development and the eventually implemented system shall conform to them. Models can be used in many phases of the development including requirements, architecture, detailed design, implementation, and testing. In continuous MBD, we consider the agile development of models in each of these phases. We offset these short development cycles with long and gated development following the V-model [26].

For example, practitioners often use diagrammatic models for their software designs [32]. Moreover, these design models may be supplemented with implementation models, from which code is automatically generated. In addition, MBD also includes those practices in which models are used for a high-level system design that is later implemented manually. Continuous MBD refers to settings in which these models are continuously developed iteratively and maintained throughout the development life-cycle. In contrast, in a non-continuous MBD setting, the high-level system design would be made upfront and then frozen so that the next phase of development can work with it.

Automated support to manage the consistency, among models or between models and other development artifacts, is vital throughout the development and (long-term) maintenance of complex software systems [27]. By consistency, we refer to the agreement and completeness of artifacts on

Communicated by K. Lano, S. Kolahdouz-Rahimi, J. Troya, and H. Alfraihi.

This work was supported by Software Center https://www.software-center.se.

✉ Robbert Jongeling
robbert.jongeling@mdu.se

Federico Ciccozzi
federico.ciccozzi@mdu.se

Jan Carlson
jan.carlson@mdu.se

Antonio Cicchetti
antonio.cicchetti@mdu.se

[1] Department of Innovation, Design and Engineering (IDT), Mälardalen University, Västerås, Sweden

Published online: 13 April 2022

aspects of the system they describe, i.e., consistent artifacts do not contradict each other and describe those parts of the system they are expected to (for example, we might expect an implementation to refine certain information from an architecture model). In this context, one of the most important impediments to adopting continuous MBD practices is the lack of support for consistency management (i.e., consistency checking and repairing) [28]. The literature is rich in approaches for tackling various MBD challenges related to consistency management, e.g., bidirectional transformations between models [24], model-metamodel co-evolution [13], and rule-based consistency checking between models [16]. But, in our industrial collaborations, we observe that it is not straightforward to adopt these approaches as part of the state of the practice. The challenges are multiple: consistency management tasks are often not prioritized; also, commonly large amounts of legacy projects need to be maintained; moreover, there are usually many people and processes involved that cannot be changed overnight. Hence, our goal is to aid companies in migrating from their current states of practice toward practices in which they benefit more from the modeling activities they do, at first to support more automated consistency management, and ultimately, to enable more continuous MBD.

To be able to provide the companies with a path toward continuous MBD, we need to understand their current states of practice and envisioned goals. We formulate the following overarching research goal:

RG Our goal is to identify the main characteristics of consistency management challenges encountered when adopting continuous MBD in industrial settings.

In particular, we focus on identifying the current state of practice of consistency management and desired future directions. We formulate the following research questions:

RQ1 Which problems related to inconsistency between development artifacts do companies experience in their MBD practices?
RQ2 How do they currently handle consistency management?
RQ3 Which future achievements do they envision for their consistency management practices?

To answer the research questions, we collected the states of practice in terms of thirteen MBD settings across nine companies. The remainder of this paper is organized as follows. In Sect. 2, we provide an overview of the background and related work. In Sect. 3, we describe our research method in detail. The results in terms of the gathered industrial settings are included in Sect. 4. A discussion of the results follows in Sect. 5 and includes answers to the research questions, a comparison between the identified challenges and approaches in the literature, and threats to validity. The paper is concluded in Sect. 6.

2 Background and related work

In this section, we discuss background on model-based development (MBD) and, in particular, work toward introducing Agile MBD practices. Furthermore, we discuss consistency management based on a selection of recent secondary studies. Finally, we relate our work to other works presenting the state of practice or challenges for consistency management.

2.1 Continuous model-based development

Continuous integration has been widely adopted since it was proposed as one of the possible Agile development practices [3]. Studies of its adoption in industrial practice have highlighted significant impediments related to test automation [40], the usability of tools, and the decomposition of complex engineering tasks [35]. Consequently, introducing continuous development practices (integration, delivery, deployment) is not a one-time action but a phased process over a longer term [41]. While these studies focus on traditional software development projects and do not target MBD specifically, we can expect a similar set of challenges to pertain to transitions from gated MBD processes toward short development cycles.

Similar to the adoption of continuous development practices, also several studies have examined the adoption of MBD in industry. Our work is closely related to these, as we will see later that there is a tight coupling between adopting "more" modeling practices and advancing consistency management. A commonly recurring frustration is a lack of tool performance [6,37]. But also organizational factors related to, e.g., established processes can be impeding factors to introducing MBD [25]. In combination, these two imply difficulty in interoperability of tooling across the organization, which indeed is a factor in industrial adoption as well as a steep learning curve for MBD and its tools [32]. The aforementioned challenges are well-established across the literature and are fundamental to the adoption of MBD.

There are few reports of industrial experiences of combining agile methods and modeling [1]. A few studies have shown early-stage benefits from agile MBD, but with few details on how continuous development practices are established [1,17]. Some experiences of introducing Continuous Integration (CI) in an MBD project have shown several hurdles to overcome, e.g., the challenge of model differencing and merging [20]. A top-down analysis of agile model-based systems engineering identifies consistency management among several challenges and proposes

lightweight, incremental, approaches to it [14]. Indeed these reported challenges are among the *usual suspects* of challenges of MBD adoption that are still relevant today [11].

In general, adopting continuous development practices is challenging because of the need to automate many manual steps involved in complex development settings. A study of adopting CI in the automotive domain notes that a lack of interoperability causes manual actions for moving information between tools [31]. A conceptual approach to counteract these challenges is to integrate modeling tools into the continuous delivery pipeline, which is far from a straightforward task and would require mature tooling and model awareness throughout the pipeline [19].

In our previous work, we interviewed practitioners at three industry partners to identify impediments to introducing continuous integration in their model-based development settings [28]. One of the most pressing challenges identified was consistency management across different central development artifacts, such as system models or software models and the implementation models or code. Therefore, we focus specifically on what types of approaches to consistency management there are and how they affect the development practice.

2.2 Consistency management approaches

More automated consistency management is among the promises of model-driven engineering (MDE) [46]. There is a plethora of work on consistency management and multi-view modeling, as is evidenced by recent SLRs on these topics with little overlap: five secondary studies from the last five years [10,12,34,38,51] contain 299 unique studies out of a cumulative 306 collected primary studies. These 299 papers are only a subset of the total work on consistency management, given the specific focuses of each SLR. As identified by those secondary studies, a common weakness in the literature is a lack of alignment of proposed approaches with industrial practice. Indeed, there are few industrial evaluations of proposed approaches. It is also unclear to what extent reported challenges in the literature are, with what relative priorities, experienced in the industry today. Therefore, it is interesting to see what the required support for consistency management in various industrial settings is.

In Sect. 5, we discuss related literature for specific identified challenges. Here, we now summarize the remainder of the literature based on the previously mentioned five secondary studies in reverse chronological order of their publication.

1: A systematic literature review of cross-domain model consistency checking by model management tools [51]. This paper focuses explicitly on model management tools and consistency across models in different domains. It finds that tools can predominantly check interface consistency. Additionally, the authors identify challenges in tool interoperability and consistency maintenance. The paper also acknowledges that no single solution will serve all purposes, as is natural given the heterogeneity of industrial practices. This observation aligns with our research direction of identifying current practices and the future direction of consistency management practices of companies.

2: Multi-view approaches for software and system modeling: a systematic literature review [12]. One of the main topics in this SLR is the consistency management between views. The paper finds that no presented multi-view approaches have been evaluated in industrial settings. Moreover, the authors identify several limitations of existing multi-view approaches. These are, among others: tool support, consistency management, versioning, distributed development, and lack of semantic consistency management.

3: A feature-based survey of model view approaches [10]. This paper shows an overview of modeling views, focusing on their provided features for synchronization, both at design time and at run-time. The authors present a feature model characterizing model view approaches, which gives a good perspective from the literature for knowing what type of mechanisms exist and what features they present. A limitation of the paper is that it does not provide much insight into what industrial use cases these approaches aim to address.

4: Systematic review of software behavioral model consistency checking [38]. This paper focuses on behavioral models, providing an overview of challenges and proposed solutions, as well as studying evidence of their application in industrial settings. The paper studies what consistency problems are addressed in the literature and distinguishes between static/dynamic/simulation and horizontal/vertical consistency. Among the suggestions for future research, the authors mention the need for more rigor in industrial evaluations, since they find that the current evaluations are mostly weak.

5: Feature-based classification of bidirectional transformation approaches [34]. This paper builds a feature model of model-repair approaches. The authors focus on model repair and explicitly exclude papers that focus on "just" impact analysis, papers that avoid inconsistency by enforcing consistent states, and papers that consider other artifacts than models in their consistency management scope. Conversely, approaches with a broad application focus (on multiple types of models and inconsistencies) are included. The authors mention as an area of future work, or lack of maturity of the field, that the proposed approaches do not guarantee the correctness of the repair actions in terms of functional semantics. Moreover, they note a lack of insight into industrial adoption:

"lack of information regarding the effective implementation of the approaches" [34].

There are further secondary studies that look at consistency management specifically for UML [33,52]. These studies reveal a multitude of approaches focused primarily on managing consistency between different diagrams of UML models. While the technical specifics of these approaches may not be transferable to settings where there are multiple models rather than a single model with multiple views, there are other considerations about, e.g., definition and invocation of consistency checks that can be considered also in our settings.

A tertiary study [49] has built a feature model of concepts related to model management based on other studies presenting feature models, namely: [4,10,12,15,22,23,30,34]. Given the variety of domains and perspectives by which consistency management problems can be described, sometimes the terminology used throughout the literature can be confusing; in this respect, the feature model proposed in [49] provides a common vocabulary. We adopt such a vocabulary in this article; in particular, in Sect. 4 we use the same consistency management aspects as included in the feature model in [49] to categorize industrial settings.

2.3 Consistency management practices and challenges

In practice, complete consistency is often infeasible and undesirable because enforcing it would inhibit the software development process. Therefore, the idea of "tolerating inconsistency" was proposed [7]. That paradigm prioritizes devising methods to identify and keep track of inconsistency rather than providing means to automatically synchronize models.

Practitioners report using models as key software development artifacts for tasks such as simulation, code generation, and test case generation [32]. The same paper shows that models are also used for structural and behavioral consistency checking. Interestingly, the paper reports that not enough data was gathered to support the hypothesis that managing the consistency of models over time is challenging. We expect that this is the case, given that still nowadays, model synchronization is a commonly considered impediment to the adoption of MBD in industry [44].

Ali et al. [2] describe how it is sometimes hard to make a business case for architectural consistency because it is hard to quantify the effects of consistency management efforts and they are rarely visible for customers of the product. Another common argument is that consistency is not important, since the purpose of the model was merely to create an initial design rather than a rigid specification. In settings where consistency is regarded as important, feedback on inconsistencies during this tolerating phase was found to help developers correctly resolve the inconsistencies [29]. Besides any of such aforementioned practical challenges, there are also fundamental limitations to applying a series of bidirectional transformations to manage consistency in networks of models [48]. In summary, reported industrial settings are needed to complement the existing work reporting specific approaches to particular consistency management challenges.

3 Research method

To supplement the literature with descriptions of industrial practices and their challenges, we conduct a multiple case study in which we collect industrial MBD settings from a diversified set of companies, describe them in detail, and discuss them with respect to our research questions. In total, we describe thirteen settings across 9 companies. MBD is practiced in each setting, and we identified various ambition levels toward continuous MBD. To illustrate a broad range of practices and consequent challenges, we have not further filtered the settings to include only those with a high desire to adopt continuous MBD. Whenever we mention a "setting" originating from companies, we refer to a particular group or a small part of the company only; this also explains why we identified multiple settings in some of the companies. Due to the different collaboration nature with different companies, we use different complementary ways for collecting data as summarized in Table 1.

We describe four settings (S09–S12) based on the data gathered in workshops in our previous and ongoing research collaborations. Within these collaborations, we had several meetings where researchers and companies agreed on research questions to be studied. As part of these meetings, we gathered data on the state of practice at the involved companies. To supplement the four initial settings, we asked other partner companies to complete a questionnaire. We followed up on the results of the questionnaires with interviews with the participants. The combined input resulted in five additional settings (S01–S05) from two companies. We organized two online workshops with companies; there we gathered three additional settings (S06–S08). As a final data source, we derived an additional setting (S13) description from a self-reported case description from a company as part of another research project. To avoid misunderstandings about the collected data, we asked for feedback from the practitioners. In the case of workshops, we summarized our findings and asked for confirmation by e-mail. In the case of structured interviews, the interviewer shared his screen and together with the interviewee filled out a spreadsheet containing columns for each question in our classification scheme and rows for separate use cases.

To systematically gather data throughout the study and across the different collaborations, we used the same classi-

Table 1 Companies involved in our study listed by anonymous IDs, their domains, how we gathered the data presented in this study from them, and the IDs of the settings that were collected per company

ID	Domain	Data gathering methods	Setting IDs
C1	Tool vendor	Questionnaire and follow-up structured interview with senior architect	S01, S02, S03
C2	Automotive	Questionnaire and follow-up structured interview with company researcher	S04, S05
C3	Automotive	Online workshop with two managers and one system architect	S06, S07
C4	Automotive	Online workshop with a researcher who was embedded in the company	S08
C5	Avionics	Workshops with two system engineers	S09
C6	Industrial automation	Workshops with architect and system engineer	S10
C7	Railway	Workshops with an architect and two software engineers	S11
C8	Consumer electronics	Workshops with two architects	S12
C9	Industrial automation	Questionnaire and self-reported use case documentation	S13

fication scheme for all settings. The content of the schema is determined by our research questions. An overview of the considered dimensions of the classification scheme is shown in the feature model in Fig. 1. Some abstract features represent open questions, while the others are broken down into their possible concrete features. This breakdown was done after data gathering, upfront we only decided on the abstract features.

In particular, we gather information related to the following items:

1. Artifacts to be consistent
   (a) What type of artifacts shall be consistent with each other? (conceptually, e.g., system model and code.)
   (b) What relationship exists between those artifacts? (e.g., generated, manually implemented, or hybrid)
   (c) What specific artifacts shall be consistent? (concretely, e.g., SysML BDD and C/C++ method declarations)
2. Inconsistency problems
   (a) What kind of problems are caused by inconsistency between these types of artifacts?
   (b) What is the severity (frequency and impact) of those problems?
3. Consistency checks
   (a) What type of consistency (e.g., structural or behavioral) is desired to be checked?
   (b) What kind of consistency checks (e.g., none, implicit, manual, automated detection, automated repair suggestions, or other) are currently used in the setting, and how are they invoked?
   (c) How are the identified inconsistencies currently repaired (e.g., not, manually, hybrid, automated, or other), with what frequency, and using what tools?
4. Correspondence links
   (a) What kind of correspondence links exist between the artifacts? (e.g., implicit by name matching, explicit by maintaining a separate model, or none at all)

In the following section, we present the gathered settings.

4 Industrial settings

In this section, we present the thirteen industrial MBD settings that we gathered and that are summarized in Table 2. We categorized the collected settings based on the types of artifacts that should be kept consistent. In the remainder of the section, we discuss the following four categories:

1. Software design models—Implementation (S03, S04, S10)
2. Software design models—Tests (S01, S02, S11)
3. System/architecture models—Implementation (structure and behavior) (S06, S08, S09, S12)
4. Many-artifacts (S05, S07, S13)

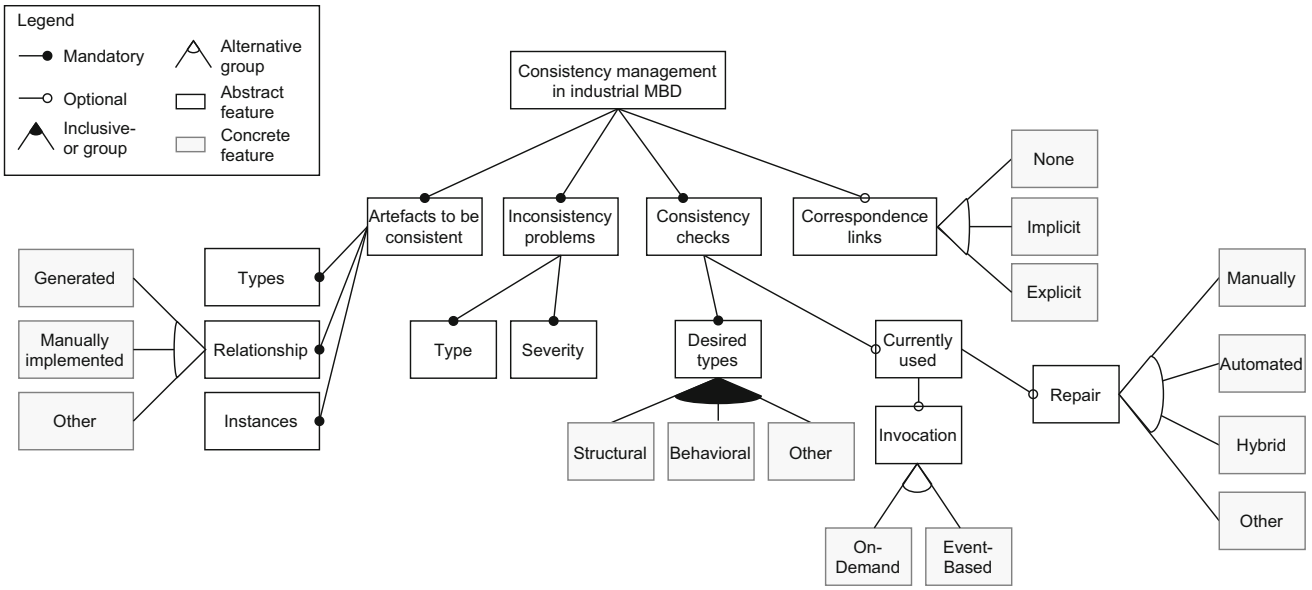

Fig. 1 The employed classification scheme, shown using feature model notation

Table 2 Summarized state of practice of modeling and continuous development in studied companies

ID	Summary of MBD practice
S01	Modeling on software level, code generation, test cases in code
S02	Modeling on software level, code generation, test cases also models (sequence diagrams)
S03	Modeling software, code generation. Continuous integration of models.
S04	Modeling implementation in DSLs, also maintaining DSL metamodel
S05	Modeling implementation in DSLs, also including many other artifacts in different formats describing aspects of the system
S06	Modeling system level and at lower levels for simulation or further decomposition
S07	Modeling system level and at lower levels for simulation or further decomposition
S08	On the way to adopting system modeling, additionally doing some simulation modeling and some Simulink for implementation (code generated). Agile development for software implementation (code+Simulink). Frequent integrations (even multiple daily).
S09	System modeling in SysML. Agile development for software implementation based on the design outlined in the system model.
S10	Modeling in in-house developed DSL with a code generator. Generated code is sometimes manually changed.
S11	Modeling only for implementation and testing, complete code generation. Moving toward adopting CI for their MBD.
S12	Modeling system arch in UML. Agile development for software implementation
S13	Part of the software is modeled in state machines (Yakindu) and code is generated from them. Other parts are manually implemented. There are also many other documents and models used to represent other parts of the system.

4.1 Software design models—Implementation

In this category, we consider those settings where models are used to describe the low-level design of software, including its structure and detailed behavior. These software design models are realized through automatic generation, manual implementation, or hybrid (a mix of the previous two). In the case of automated code generation, consistency from model to code is achieved by construction. Completely manual implementation is uncommon since it would imply redoing work already done when creating the design models. In this category, we consider specifically the hybrid form, since it triggers some interesting use cases for consistency management.

Indeed, challenges arise when automatically generated code is manually supplemented or modified. Modifications may happen to customize the code for different target applications, complete incomplete specifications, or simply because

some functionality is easier to express in a textual notation than in diagrammatic models. Typically, these problems are dealt with by generating code with protected regions, which are not overwritten upon code re-generation (as in, e.g., Epsilon Generation Language [45]). This approach is suitable in cases where the code expresses additional things that cannot be expressed in the model. In cases where the model and code must express the same information (e.g., for maintenance reasons), additional measures are needed to enable the propagation of changes from the code back to the model. We have identified three industrial use cases in this category, as described in Table 3 following the structure of the classification scheme presented in Sect. 2.

S03: In this setting, the company is generating C/C++ code from state machines expressed in UML-RT. A consistency management challenge occurs here because it is desired to treat both models and code as two views of the same artifact: not only should changes in the model be reflected in the code through re-generation, but also any change to the code should be directly and completely reflected in the model. An automated synchronization mechanism is in place that propagates changes between artifacts automatically when a change occurs; the mechanism is based on protected regions.

S04: In this setting, the company uses several powerful MDE concepts for developing Domain-Specific Languages (DSLs) and automatically generating code from models conforming to those DSLs. Models conforming to these DSLs are interrelated through dependencies and are therefore in need of maintenance upon evolution of the DSLs. Indeed, the main consistency challenge here relates to the problem of metamodel and model co-evolution [13]. Specifically, an update to one DSL can result in non-resolvable references in a model conforming to another DSL. Inconsistencies in this regard are very impactful since they prevent building and running the software. Currently, the company has some automatic checks to detect such reference issues. Resolving the inconsistency is currently a manual task for the developer, but this might be enhanced with suggestions for fixes in the future.

S10: In this setting, the company is generating PLC (programmable logic controller) code from a model expressing a functionality. The model is created in a DSL that is tightly integrated into a modeling tool, both developed in-house. The generated code is a combination of automatically generated instructions from the model, libraries, and other static pieces of code.

To customize this code for different operational environments, the generated code, rather than the model, is modified, leading to inconsistency between the code and the model. This inconsistency mainly impacts the maintainability of the code, since the models can no longer be used as accurate documentation of the implementation. Hence, the company aims to re-establish consistency between the models and the code, but this is complicated by the manual changes to the generated code. Currently, consistency is restored manually, at fixed steps in the development process. In the future, the company aims at a more continuous feedback loop and more frequent checks of the consistency between model and code.

4.2 Software design models—Tests

In this category, we consider test cases and their consistency with the model or code under test. When tests are consistent with the design models, it is meant that not only the test cases pass, but also that the test cases appropriately cover the models, i.e., test the correct things. When a system evolves, challenges arise. Inconsistency caused by, e.g., a drop in test coverage is much more subtle to detect than test case failure. We found challenges in keeping test cases and implementation models consistent in the three industrial use cases summarized in Table 4. Here, the consistency checks do not refer to running the test cases but instead refer to (currently manual) alignment checks of test cases and the implementation.

S01 and S02: In these settings, models are tested, and then code is generated. The tests can be expressed in either test scripts (e.g., Python) or as models (e.g., UML state machines). The main consistency management challenge in this setting relates to co-evolving tests with the model. Again, when the tests fail, their need for co-evolution is obvious, but in other cases, this need is less evident. The danger of inconsistent artifacts here is that the tests become meaningless if they do not cover the models appropriately. Currently, manual assessment of consistency and handling of repair actions is required.

S11: In this setting, consistency between tests and model under test is desired. Models are used for complete code generation (unlike in S03, here the code is not itself modified). Engineers spend significant manual effort on analyzing the impact of changes to these models on their test cases. In particular, it must be analyzed if the tests still cover the implementation models correctly and completely. Moreover, in this setting variants created through a clone-and-own method [18] need to be considered too. One of the consequences is that there may be significant overhead needed to analyze existing test models and assess to what extent they are appropriate for a newly branched-off product variant too.

Development cycles could be made shorter if there were more support for managing the consistency between the evolving models and tests, and hence to reduce manual analysis effort. One direction that could help this setting is to migrate from clone-and-own to more structured re-use following the product line engineering paradigm. Nevertheless, even with adoption of the best practices of that paradigm, there remain a need for impact analysis to assess the effects of a changes model on the test cases. Hence, we foresee that in the future, this setting could be enhanced by auto-

Table 3 Category 1: Software design models—Implementation. Industrial settings of MBD where consistency management between software design models and their implementation is desired

Setting ID	Conceptual artifacts consistent	Relation artifacts	Inconsistency problems	Concrete artifacts consistent	Consistency type	Correspondence links	Consistency checks (invocation mechanism)	Repair actions
S03	Model—Manually modified generated code	Automatically generated code, but manually modified	Model and code must be identical to allow mixed editing of notations	UML-RT models—C++ code	Equivalence	Through protected regions	Does not apply, see repair actions	Automated, action-triggered, synchroniza-tion
S04	models of DSLs	Linked through usages, imports, etc.	Update to one DSL might cause errors in another DSL that depends on concepts. (co-evolution-like)	Models of in-house developed and maintained DSLs	Complete syn-chronization might not be possible, but at least changes that impact dependent artifacts are synchronized.	Explicit dependencies between models	Automatic, for some type of structural consistency, raises a kind of syntax error if something is referenced that is not there.	Manual. In future perhaps fix suggestions
S10	Model—Manually modified generated code	Automatically generated code, appended with libraries and other code not linked to the model, and moreover, manually modified	Model must reflect code even after changes, for maintenance operations	Model in a DSML—generated IEC61131-3 code	Structural and behavioral equivalence of parts of the code dictated by the model	Implicit (through code generation)	Manual, process-triggered	Manual repair, whenever inconsistency has been identified

Table 4 Category 2: Software design models—Tests. Industrial settings of MBD where consistency management between design models and tests is desired

Setting ID	Conceptual artifacts consistent	Relation artifacts	Inconsistency problems	Concrete artifacts consistent	Consistency type	Correspondence links	Consistency checks (invocation mechanism)	Repair actions
S01	Model—Test cases	Manually created and maintained test cases, testing implementation as in models	Test cases may become inconsistent when model evolves	UML-RT models—Test scripts	Behavioral correspondence (tests must cover the implementation)	Explicit links from test scripts to model under test	Manual, on demand	Manual, on demand
S02	Model—Test cases	Manually created and maintained test cases, testing implementation as in models	Test cases may become inconsistent when model evolves	UML-RT models—Tests in state machines	Behavioral correspondence (tests must cover the implementation)	Explicit links from test model to model under test	Manual, on demand	Manual, on demand
S11	Model—Test cases	Manually created and maintained test cases, testing implementation as in models	Test cases may become inconsistent when model evolves or new model variant is defined	Simulink models—Simulink test models	Behavioral correspondence (tests must cover the implementation)	Explicit links from test model to model under test	Manual, on demand	Manual, on demand

mated support for the co-evolution of the test cases. Given the complexity of co-evolution in the general case, this support would most likely consist of repair hints, rather than automated repair actions.

4.3 System/Architecture models—Implementation

In this category, we consider models at a higher level of abstraction than those discussed in the previous categories. Both the software aspect of system models and architecture models typically focus on describing the distribution of functionality over components of the system and the interactions between them. From system and architectural models, there is no possibility to generate code since they are not intended to (and therefore do not) describe the detailed behavior of the components. Nevertheless, consistency between these models and code is still imperative for correct development and manageable maintenance of the system. An overview of the industrial settings in this category is shown in Table 5.

S06: In this setting, high-level models are mostly means to understand what needs to be built. For some parts, a decomposition is made into software components, which are modeled too. The eventual implementation is done manually, based on these models. Since the systems under development are usually very large, requirements are captured in some of these models to keep the engineering process manageable. It is accepted by engineers that the models are not fully consistent with the eventual implementation, so the initial impact of an inconsistency is small. Nevertheless, it is required that the eventual implementation conforms to the requirements. In the current setting, there is no feedback loop between the model and implementation and consequently, the software engineers lack insight into the bigger picture that the model provides. An adjacent challenge is that the scope of modeling is not well defined in the company, causing uncertainties about what must be modeled at what level of abstraction. Currently, there is no automated support to check consistency between the model and the implementation, nor is there any support for repair actions.

S08: The company is working toward adopting model-based systems engineering. In this setting, the rationale for dependencies between separate subsystems is captured in a SysML model. Moreover, the implementation of these subsystems is modeled in Simulink, from which code is generated. The main goal of modeling in this setting is to document design decisions. This makes it slightly different from the other settings, where models play a more central role and are used for the development itself. Nevertheless, also in this setting consistency between system (SysML) and implementation models (Simulink) must eventually be kept intact; otherwise, the documentation becomes unreliable.

The link to continuous MBD in this setting is less clear since the aim of modeling in the first place is a bit questionable. It would be more beneficial to, e.g., generate a skeleton model from the SysML model that then can be completed by Simulink models.

S09: In this setting, the implementation is done manually in C++ based on a high-level description of components and their interfaces in a SysML system model. In this setting, the system model does not contain many low-level behavioral details of the software parts; therefore, code cannot be generated automatically. Rather, the model focuses on dividing the functionality of the system over multiple software components and assigning them to different hardware components. To enable long-term maintenance of these complex systems, consistency between model and implementation is imperative. On the one hand, the implementation must follow the design as outlined in the system model. On the other hand, the system model must provide a reliable image of the actual implementation. Moreover, the company aims to reuse components across multiple products, which requires the models and code to be synchronized.

During development, incremental changes of the system are performed on the model and communicated in small chunks to the software engineers who then implement the new or changed functionality. This manual implementation and the communication overhead deriving from it inhibit to some extent the introduction of more continuous development practices. Consistency checks between model and code could help by indicating to what extent the model and code agree, thereby decreasing the amount of manual effort currently spent on consistency checking and impact analysis. Indeed, the company has set the goal to provide system and software engineers with support for synchronization of model and code. In particular, they envision a bridge that enables automated analysis of the completeness and consistency of the implementation as compared to the model, and vice versa. The main challenges toward this goal are to align syntactic and semantic differences between model elements and their corresponding code elements.

S12: In this setting, the system architecture is modeled using UML diagrams that describe the system's components, interfaces, and connections between them. Some of the components are implemented in-house and others are open-source components. Consistency management is achieved through a set of scripts that check the code for structural consistency, particularly focusing on interface violations. The main challenge for consistency management in this setting is to deal with different variants and versions of both software components and the system. For example, it may happen that until version n of the system, component `X` is used, but from version $n+1$ onward, `X` is replaced by `Y` and `Z`. Moreover, each component may exist in different versions through time, e.g., releases of open source components. Accurately reflecting this in the model is rather challenging and using

Table 5 Category 3: System/Architecture models—Implementation. Industrial settings of MBD where consistency management between high level models and implementation is desired

Setting ID	Conceptual artifacts consistent	Relation artifacts	Inconsistency problems	Concrete artifacts consistent	Consistency type	Correspondence links	Consistency checks (invocation mechanism)	Repair actions
S06	System model–SW models–implementation	SW models decomposition of system model functions. Implementation further detailed realization of the requirements from the system model.	Not that much, inconsistency is omnipresent but most of the time not really a problem.	SystemWeaver models—C/C++ code	Behavioral	Some degree of implicit naming equivalence between system model and implementation	None currently	None currently
S08	System model—Executable model	Manually implemented executable model, system model as documentation	Not that much SysML being used for documenting rationale for dependencies between parts of the system	SysML model—Simulink models	Structural	Implicit, through name equivalence	None currently	None currently
S09	System model—Code	Manually implemented code, based on system model	Model must be a reliable representation of the code, and vice versa	SysML structure diagrams—C++ code	Both structural and behavioral consistency (for as far as it is included in the system model)	Implicit through name similarity	Manual, on demand, e.g., in code reviews	Manual, on demand
S12	Architecture model—Code	Manually implemented code, model just prescribes components, interfaces, and configuration	It must be ensured that code does not violate interfaces as defined in architecture	UML component model—C++ code	Structural	Implicit, through name equivalence	Automated, script ran on demand, reports violations of interfaces	Manual, on demand

that information as input for the consistency checks already in place is challenging too.

When in place, consistency checks between all variants and versions of the architecture model and the code will allow for better management of the software components (both those developed internally and open-source components on which the system relies). The company has set the goal of explicitly modeling variants and versions in their architecture model and benefiting from that information through automated consistency checking mechanisms. It is worth noting that these needs exceed standard variability management due to the additional dimensions that need to be incorporated in the model, including the variants and revisions of components as well as of the system itself.

4.4 Many-artifacts

In this category, we consider those settings that require consistency among three or more development artifacts, ranging from requirements to implementation. An overview of the industrial settings in this category is shown in Table 6.

S05: In this setting, the main consistency management problems are related to the late detection of contradictions between different complementing artifacts. Some of the artifacts need to be consistent for accurate implementation, such as requirements and integration tests. Other artifacts must be consistent for system maintenance activities such as step-by-step guidance (commands) and root cause analysis (sensor values and formulas). The different data sources containing the artifacts are not always models but can also reside in unstructured formats such as spreadsheets and text documents. The automated support for consistency checking is limited to a few types of models, that, when loaded, reuse information from the shared knowledge space. When these checks fail, the engineers have some help with repair actions from visualizations indicating where the inconsistency is. Other inconsistencies are detected manually, increasing the risk of them going undetected. Moreover, these manual activities mean that only a few pre-defined types of inconsistencies are checked. In addition to what was mentioned in S04, here we also consider requirements specified in natural text, other artifacts describing the intended functionality of the system, and integration tests. These additional artifacts give consistency management a different focus. The company is currently working on establishing consistency checks via a normalized format for several artifacts. This work entails extracting data from the different tools, representing it in a common format, running consistency checks across the normalized representations, and finally visualizing the results.

S07: In this setting, as in S06, system modeling is used as a way to capture requirements and create a first decomposition of the system. Within this process, consistency management between several different artifacts is relevant, including system model, AUTOSAR models, code, and tests. A portion of the system model is refined in AUTOSAR models, from which code is generated. The other portion of the system model is implemented manually. Similar to S06, inevitably inconsistencies occur between these different representations, but usually they are not a problem because the engineers are typically aware of them. The engineers do not update the artifacts once an inconsistency is introduced. The only imperative consistency is the behavioral consistency between requirements and the eventual implementation, since the impact of inconsistencies between these artifacts is the largest. There are explicit consistency links only between requirements in the system model and test cases and they are defined by annotating the latter ones. Currently, the engineers perform manual consistency checks rather late in the process, any earlier noticed inconsistencies are ignored until related problems arise and also then repair actions are performed (manually). The future direction for the company in this context is to enhance the consistency check by writing more detailed requirements such that it will be easier to define test cases that show inconsistency between requirements and implementation.

S13: In this setting, there are several types of artifacts involved: requirements, software design models, and a mix of automatically generated and manually implemented code. Moreover, there may be other relevant artifacts such as CAD models that ideally should be kept consistent with the others too. The engineers identify a need for reliable tools that track different artifacts and do not hinder the existing workflow. Among the reasons for this need of tracking different artifacts is that the company uses a wide range of different modeling languages and tools across different projects. For this reason, it is complicated for engineers to switch between projects. Moreover, even within a single project, the diversity of tools leads to complex documentation, which is overwhelmingly written manually. One of the reported problems with respect to consistency management is that miscommunication occurs between different people working in different phases of the same project. The severity of inconsistencies here grows proportionally to the system itself; in small projects, the inconsistencies can be managed by brief discussions between engineers, but in large projects resolving inconsistencies requires long analysis. Most probably, this is because there is a lot of room for interpretation in each step of the modeling process, given the plethora of modeling formalisms and tools used, and also because there are no explicit links between these models. The company found it difficult to concretize future directions related to consistency management. The main intention is to somehow improve the current disjoint state of all the models.

Table 6 Category 4: Many-artifacts. Industrial settings of MBD where consistency management between more than two artifacts is desired

Setting ID	Conceptual artifacts consistent	Relation artifacts	Inconsistency problems	Concrete artifacts consistent	Consistency type	Correspondence links	Consistency checks (invocation mechanism)	Repair actions
S05	Almost everything: requirements, parts lists, measurement channels, error handling, commands, integration tests, sensor values and formulas.	Different data sources are complementing each other, but are in widely different formats	Many, in worst case rather late detection.	Textual requirements—parts lists measurement channels—error handling—commands—integration tests—sensor values	Contradictions between the artifacts should be avoided	Through device knowledge space	When a model is loaded and reuses info. from the shared knowledge space	Manual, helped by some visualizations
S07	System model–SW models–implementation	SW models decomposition of system model functions. Implementation further detailed realization of the requirements from the system model.	Sometimes problems can occur when requirements not updated after code changes	SystemWeaver—AUTOSAR—C/C++ tests	Behavioral	Explicit between requirements in system model and test cases	Manual, during integration and validation stage	Manual
S13	Requirements, Architecture/design, State machines, code, tests	Hybrid	Several w.r.t. miscommunication, complicated documentation and a steep learning curve for people new to a project to become productive	Natural text requirements—Architecture/design sketches in DrawIO—Yakindu State machines—C/C++ code, tests	All types	None	None	None

5 Discussion

In this section, we analyze our findings described in Sect. 4 and answer our research questions. Moreover, we briefly indicate support in the literature for our categories of industrial MBD settings. We also reflect on the employed classification scheme and its use for answering the research questions. Lastly, we discuss threats to validity.

5.1 Consistency management characteristics

In this subsection, we summarize our findings presented in Sect. 4 to answer the three research questions.

5.1.1 Encountered problems due to inconsistency (RQ1)

There are settings, such as S06, S07, and S08, where consistency is not considered important. Usually, engineers in these settings indicate that they use models merely for communicating initial ideas and then abandon them. Hence, there is no interest in keeping them updated and they are most probably going to be inconsistent with the eventual implementation.

In other settings, such as S01, S02, S04, S11, and S12, inconsistencies are important since multiple development artifacts are tightly connected. In S01, S02, and S11, test cases must co-evolve with the models such that they remain up-to-date. In S04 and S12, inconsistencies can cascade to other artifacts, but the problems related to inconsistency are similar; it must be ensured that the closely related artifacts do not contradict each other, such that the dependencies are not violated and new changes do not rely on outdated information.

In the remaining settings, S03, S09, and S10, inconsistency problems are related to other aspects of maintainability, particularly the correctness of models with respect to their intended use. In S03, model and code represent two views of the same artifact and must therefore be kept consistent. In S09 and S10, there is a larger conceptual gap between model and code. The model is used both for design and, differently from those settings where consistency is not important, it is used for documentation too. To ensure, on the one hand, that the implementation conforms to the design and, on the other hand, that the model accurately reflects the implementation, inconsistencies between them need to be resolved. In these settings, the model is typically ahead of the code, i.e., it prescribed the future, whereas the code represents the current status of the implementation. Therefore, it is not necessarily useful to be constantly reminded of inconsistencies between the artifacts, but it is crucial to not keep track of them over time.

> In summary, we answer RQ1 by identifying three types of encountered problems due to inconsistency: one in which inconsistency causes no or minimal problems, one in which inconsistency causes cascading contradictions between dependent artifacts and one in which inconsistency causes incorrect models.

5.1.2 Handling of problems related to inconsistency (RQ2)

We noticed the trend across the studied settings that the bigger the conceptual gap between the artifacts, the fewer explicit correspondence links are in place. In a majority of settings (S01, S02, S06-S09, S11, S12), there is a reliance on name equivalence or name similarity for matching parts of the model to parts of the implementation. The risk of this practice is that while engineers might be using code or modeling guidelines to maintain these mappings, they are very easily eroded throughout development. Another consequence of the bigger conceptual gap is that the type of consistency that can be checked is usually only structural, simply because the high-level model does not (and should not) contain the amount of detail on the behavior that would be necessary to check the behavioral consistency between the model and its implementation. Indeed, the consistency types of interest depend on the amount of detail modeled. In general, we can notice the trend that the larger the conceptual gap, the vaguer and more focused on structure the possible consistency checks become.

Across the studied settings, there is often no support or only implicit manual support for consistency checking (e.g., during code reviews in S06 or during integration and validation in S10). Therefore, also the execution of the consistency checks is mostly done manually and on-demand, while only a few settings (S02, S07, S11) adopt automatically triggered consistency checks. Automated repair actions were found only in S02, other settings did not have them, probably as a result of the typically large conceptual gap between artifacts that makes automated repair impossible. In a majority of the settings, inconsistencies are repaired manually and on-demand. In settings S08, S09, and S13, there is currently no work on repairing inconsistencies.

Problems related to inconsistency are hence mostly handled in an ad-hoc manner. We believe that senior engineers can initially keep track of important required consistencies across specific artifacts, however in later stages, a mental map no longer suffices, and the need for more structured and automated approaches to consistency management arise.

In summary, we answer RQ2 by observing that overall, while multiple companies have plans to invest in improving their practices, these are often pushed back by more immediate concerns. When handled, a majority of settings relies on implicit correspondence links such as name equivalence. As a consequence of the typically large semantic gap between the model and other artifacts, consistency repair is almost always manual.

5.1.3 Identified future directions for consistency management (RQ3)

Tables 3, 4, 5, and 6 reflect the current state of practice. During the data collection, we also collected data to reflect the future desired state of practice in the companies, following the same classification scheme as for the current state of practice. The developed artifacts, their relation, the desired consistency, and associated problems, remain the same in all settings. Since a lot of the data are repeated, we do not include additional tables for them but instead provide a summary here.

The companies indicated little ambition in improving the traceability across artifacts, with only one indicating the desire to adopt more correspondence links that will still rely on name equivalence. For consistency checks, in a majority of settings companies indicated the desire to adopt some form of automated detection. This was expected to help both to prevent cascading contradictions and to improve maintainability (see RQ2). In several settings, the desire was to run the automated consistency checks as soon as possible, before cascading effects. For example, in S01 and S02, the test cases or a selection of them should be executed as soon as possible after a change to the model. In essence, the practitioners in these settings want to go toward continuous MBD, where changes to the model are continuously validated before contradictions may propagate elsewhere. For repair actions, the engineers agreed that large conceptual gaps between artifacts prevent automated repair. Hence, the companies did not aim for this.

In summary, we answer RQ3 by noting that overall, future directions for consistency management indicated by the companies are limited to slightly more automated consistency checks. The primary goal of the companies is to get an enhanced insight into their various development artifacts. More advanced actions such as automated repair are typically not planned.

5.2 Linking consistency management and continuous MBD

The benefits of involving high-level models in continuous development include the possibility of continuous architectural consistency checking as well as continuous traceability between high-level models and implementation. The former is particularly relevant in modern software engineering projects, where the architecture is moving toward being more dynamically evolving, rather than being fixed upfront. The latter is relevant in any setting but becomes more pressing in continuous development settings due to the need for, e.g., fast change impact analysis for changes in the high-level models.

The implementation in the studied settings is typically done in short cycles, even if the complete development is not. Depending on the setting, there are different non-functional impediments to achieving shorter development cycles for also the higher-level models. This can be related to a lack of traceability between model and code and the corresponding difficulties in impact analysis (S09) or test changes (S11). In other settings, objections are raised to continuous development involving higher-level models. In particular, practitioners fear that the current practices will not scale due to manual actions in the current process that would form bottlenecks when done in shorter cycles.

There is a tight relation between the two topics: consistency management is a significant hurdle to jump toward more continuous MBD. In almost all the studied settings, it is clear that 1) the companies are struggling with manual actions related to various forms of consistency management that impede shorter development cycles, and 2) the companies are trying to automate parts of their consistency management activities as a step toward shorter development cycles and indeed continuous MBD.

Among the reasons for the non-alignment between high-level and low-level models is that they are typically developed by different people, in different teams, with different priorities. For example, practitioners (in S06 and S07) identified a lack of willingness of developers to model: "*developers will not look in the model if they can avoid it, nor propagate changes back to model*." Consequently, the short development cycles are limited to the implementation, and consistency with the high-level model is not prioritized.

The current state of practice we encountered in the companies does not utilize powerful modeling mechanisms from the MDE paradigm, which would theoretically provide a lot of the desired consistency management "out of the box." It seems that the amount of modeled information is typically limited and a move to more comprehensive modeling approaches is not appealing. Sometimes, the modeling is done as just one way of capturing some design decisions, e.g., on the system decomposition or the architecture, however, the companies are not benefiting from the additional

semantics of expressing this in a *model*. Instead, the models become somewhat interchangeable artifacts, i.e., from the context of consistency management, it would make no difference if a document-based system would be used for what is now modeled. There is an opportunity to take more advantage of the precise semantics that a modeling language provides.

The degree to which continuous development practices are used varies greatly among the settings. In most cases, short development cycles are used only in the scope of the implementation, probably because in that area there is a lot of existing process and tool support for Agile development. Usually, the design or architecture models of a system are not included in these short development cycles, despite the potential benefits of doing so. Overall, the companies employing fewer modeling languages and tools (S01-S05 and S11) can work in shorter development cycles, due to a simplified synchronization and tool interoperability effort. Conversely, a large gap between the models implies a largely manual effort toward synchronization, which impedes the adoption of shorter development cycles.

> A reflection on our research goal *"To identify the main characteristics of consistency management challenges encountered when adopting continuous MBD in industrial settings."*: In summary, companies are struggling with their paths towards adopting more continuous MBD. Managing consistency is complicated and with more involved artifacts, this complexity grows. Continuous development is typically limited to the implementation and does not usually involve design or architecture models. Time-consuming manual consistency management activities impede the inclusion of these higher-level models into the continuous development cycle.

5.3 Approaches in the literature and identified gaps

We now briefly reflect on the support in the literature for the categories of consistency management settings we identified in this paper. In summary, we find that in most cases approaches are available for dealing with the industrial challenges, but it is still complicated to migrate from an existing state of practice to one in which these approaches are adopted.

Software models—Implementation The challenges in the described industrial settings S03 and S04 relate to the well-studied concepts of model-code round-trip engineering (e.g., [39]) and model-and-metamodel co-evolution (e.g., [13]), respectively. S10 is less well supported by existing approaches. Within the MDE paradigm, synchronizing model and code is typically done using bidirectional transformations (e.g., via triple-graph grammars [21]). We encountered limitations of these approaches in S10 because there, not all changes to the generated code should be propagated back to the model. In those cases, more control is needed on what changes should be propagated back and what changes should be ignored.

Software design models—Test A study on co-evolution of test and production code found a mix of phased and continuous co-evolution depending on the followed development paradigm [53,54]. There have been studies looking at specific instances of co-evolution in MBD, such as the co-evolution of Simulink models and their test cases [43]. In our studied settings, the main manual effort in evolving the test cases effort is spent on performing a change impact analysis.

System/Architecture models—Implementation A challenge for model transformation approaches occurs when the gap between the high-level and low-level models cannot be bridged by automated means. It is hard to quantify the effects of architectural consistency and these effects are rarely visible to customers [2]. This is in line with our observation in the introduction that consistency management tasks are not often prioritized. At the same time, the main obstacles to the adoption of consistency management approaches are the too high expected cost and effort [50]. A more lightweight approach is therefore required. Moreover, in these settings, it should be possible to benefit from the more semantically defined SysML model compared to sketched models.

The key element in existing approaches is the need to bridge the semantic gap between architecture and implementation. As found also in other works on that gap: "current approaches cannot handle the synchronization when there is a significant abstraction gap between architecture and code" [42]. A realistic example setting is S09, where work is needed on synchronizing a system model in SysML and its corresponding implementation in code.

Many-artifacts Atkinson and Stoll propose projective modeling in the form of a single underlying model (SUM), where all other artifacts are views on that SUM [5]. Consistency is then guaranteed by construction since any edit to any view is a change to the SUM and since other views are projections of that SUM, the change is immediately propagated to them. This approach requires a unified metamodel of all artifacts, which is not always possible, especially not when free-form artifacts such as textual requirements are included (e.g., in S13). Moreover, code is usually not included in these kinds of many-view management settings, probably due to the abstraction gap between implementation and high-level models. The industrial settings could benefit from approaches that provide insight into a large number of structured and unstructured development artifacts. In this context, *megamodels* have been proposed for providing a kind of floor plan of a setting, denoting all modeling artifacts the relations and transformations between them [9]. The basic idea of megamodels can

be extended to be a framework supporting traceability and consistency across model artifacts [47].

5.4 Assessment of classification scheme

We briefly reflect on the data gathering phase and specifically on the employed classification scheme. In the data gathering phase, we noticed that, given only the classification scheme, industrial partners were not able to easily answer our questions. In fact, it was often needed to have a general conversation about the involved development artifacts and the relationships between them. The classification scheme was then very useful for mapping the open answers to a smaller and more structured set of categories. Initially, we also aimed at gathering, as part of the classification scheme, the challenges that the companies themselves identify in migrating toward better consistency management and continuous MBD. The experience of data gathering showed that it was more effective to derive those challenges during the analysis since they were hard for practitioners to formulate during conversations.

5.5 Threats to validity

A threat to the internal validity of the study is our sampling of industrial settings and the ways we gathered information from them. In the end, our sample is still a convenience sample, we included all settings from our research collaborations that to some degree are practicing MBD. On the positive side, we have gathered a broad range of varying settings. On the other hand, as a consequence of this broad range of companies and settings, we could not employ the same research method for gathering information from all the companies. To mitigate this risk, we have adhered to the same classification scheme for the data, regardless of how the information was gathered. Moreover, in each of the settings, we have allowed the companies to provide feedback on the filled-out classification scheme for their settings. We predetermined the classification scheme before doing the study, to avoid bias toward certain settings. In particular, the classification is based on our research questions and the terminology used in the literature.

A threat to external validity is that our results do not generalize beyond the studied settings. To limit this threat, we considered a variety of settings while including not too many from the same companies. Moreover, we excluded those settings in which there was no MBD practice. In any case, our study cannot provide a complete overview of the state of practice and that was also not our goal. Instead, we argue that if we encounter these settings at a few companies, it is likely that there are more industrial settings similar to them and therefore, our findings can contribute to providing researchers an insight into the state of practice of model-based development in current industrial settings.

6 Conclusions

In this paper, we studied thirteen MBD settings from nine companies and analyzed them concerning continuous development and consistency management practices. One of the main threads in this paper is that we consider consistency management as an essential ingredient for shorter development cycles. Manual actions related to consistency management are impeding shorter development cycles, and therefore, the companies generally aim to somewhat automated consistency checks. To achieve more automation, the companies need to make more use of the semantics of models as development artifacts, instead of their current use, where their use is often limited and may in some cases be considered interchangeable with textual descriptions. To move toward continuous MBD in each different setting, we need to close the gap between the development artifacts so that traceability and consistency management can be supported with automation. Moreover, when adopting new development practices, we need to take into account the existing development artifacts, complex development processes, and a large number of involved engineers. Hence we identify the need for researching, in addition to new approaches, also migration paths from the current states of practice toward ones in which these approaches are adopted. Our maturity model shows high-level steps that must be taken from various starting points.

We hope that these results give researchers insights into the state of practice and encountered consistency management challenges. Our analysis can help practitioners identify their own state of practice better and help them to understand the consistency challenges toward developing in shorter development cycles. Moreover, our analysis can be of interest to practitioners who are not currently working with consistency management by identifying how they can benefit from doing so.

Funding Open access funding provided by Málardalen University

References

1. Alfraihi, H., Lano, K.: The integration of agile development and model driven development—A systematic literature review. In: MODELSWARD, pp. 451–458 (2017). https://doi.org/10.5220/0006207004510458
2. Ali, N., Baker, S., O'Crowley, R., Herold, S., Buckley, J.: Architecture consistency: state of the practice, challenges and requirements. Empir. Softw. Eng. **23**(1), 224–258 (2018). https://doi.org/10.1007/s10664-017-9515-3
3. Ambler, S.: Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. John Wiley & Sons, Hoboken (2002)
4. Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H.S., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., Zündorf, A.: Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw. Syst. Model. (2019). https://doi.org/10.1007/s10270-019-00752-x
5. Atkinson, C., Stoll, D.: Orthographic modeling environment. In: International conference on fundamental approaches to software engineering, pp. 93–96. Springer (2008). https://doi.org/10.1007/978-3-540-78743-3_7
6. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context—Motorola Case Study. In: LNCS 3713, pp. 476–491. Springer (2005). https://doi.org/10.1007/11557432_36
7. Balzer, R.: Tolerating inconsistency. In: proceedings of the 13th international conference on Software engineering, pp. 158–165. IEEE Computer Society Press (1991)
8. Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al.: Manifesto for Agile Software Development (2001). URL http://agilemanifesto.org
9. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: proceedings of the OOPSLA/GPCE: best practices for model-driven software development workshop, 19th Annual ACM conference on object-oriented programming, systems, languages, and applications, pp. 1–9. Citeseer (2004)
10. Bruneliere, H., Burger, E., Cabot, J., Wimmer, M.: A feature-based survey of model view approaches. Softw. Syst. Model. **18**(3), 1931–1952 (2019). https://doi.org/10.1007/s10270-017-0622-9
11. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Softw. Syst. Model. **19**(1), 5–13 (2020). https://doi.org/10.1007/s10270-019-00773-6
12. Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Multi-view approaches for software and system modelling: a systematic literature review. Softw. Syst. Model. **18**(6), 3207–3233 (2019). https://doi.org/10.1007/s10270-018-00713-w
13. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 2008 12th International IEEE enterprise distributed object computing conference, pp. 222–231. IEEE (2008). https://doi.org/10.1109/EDOC.2008.44
14. Denil, J., Salay, R., Paredis, C., Vangheluwe, H.: Towards Agile Model-based Systems Engineering. In: MODELS (Satellite Events), pp. 424–429 (2017)
15. Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K.: A three-dimensional taxonomy for bidirectional model synchronization. J. Syst. Softw. **111**, 298–322 (2016). https://doi.org/10.1016/j.jss.2015.06.003
16. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. **37**(2), 188–204 (2010). https://doi.org/10.1109/TSE.2010.38
17. Eliasson, U., Heldal, R., Lantz, J., Berger, C.: Agile model-driven engineering in mechatronic systems—An industrial case study. In: International conference on model driven engineering languages and systems, pp. 433–449. Springer (2014). https://doi.org/10.1007/978-3-319-11653-2_27
18. Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Enhancing clone-and-own with systematic reuse for developing software variants. In: 2014 IEEE International conference on software maintenance and evolution, pp. 391–400. IEEE (2014). https://doi.org/10.1109/ICSME.2014.61
19. Garcia, J., Cabot, J.: 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, pp. 19–32. Springer (2018). https://doi.org/10.1007/978-3-030-06019-0_2
20. García-Díaz, V., Pascual Espada, J., Núnez-Valdéz, E.R., Pelayo, G., Bustelo, B.C., Cueva Lovelle, J.M.: Combining the continuous integration practice and the model-driven engineering approach. Comput. Inform. **35**(2), 299–337 (2016)
21. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 21–43 (2009). https://doi.org/10.1007/s10270-008-0089-9
22. Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: a survey. IEEE Trans. Softw. Eng. **43**(5), 396–414 (2016). https://doi.org/10.1109/TSE.2016.2610424
23. Hidaka, S., Tisi, M., Cabot, J., Hu, Z.: Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. **15**(3), 907–928 (2016). https://doi.org/10.1007/s10270-014-0450-0
24. Hu, Z., Schurr, A., Stevens, P., Terwilliger, J.F.: Dagstuhl seminar on bidirectional transformations (bx). ACM SIGMOD Rec. **40**(1), 35–39 (2011). https://doi.org/10.1145/2007206.2007217
25. Hutchinson, J., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Sci. Comput. Program. **89**, 144–161 (2014). https://doi.org/10.1016/j.scico.2013.03.017
26. INCOSE: Systems Engineering Handbook, v3.2.2 (2011)
27. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011(E) Systems and software engineering—Architecture description. Tech. rep., International Organization for Standardization (2011). https://doi.org/10.1109/IEEESTD.2011.6129467
28. Jongeling, R., Carlson, J., Cicchetti, A.: Impediments to introducing continuous integration for model-based development in industry. In: 2019 45th Euromicro conference on software engineering and advanced applications (SEAA), pp. 434–441. IEEE (2019). https://doi.org/10.1109/SEAA.2019.00071
29. Kanakis, G., Khelladi, D.E., Fischer, S., Tröls, M., Egyed, A.: An empirical study on the impact of inconsistency feedback during model and code co-changing. J. Object Technol. **18**(2), 10–1 (2019). https://doi.org/10.5381/jot.2019.18.2.a10
30. Knapp, A., Mossakowski, T.: Multi-view consistency in UML: A survey. In: Graph Transformation, Specifications, and Nets, pp. 37–60. Springer (2018). https://doi.org/10.1007/978-3-319-75396-6_3
31. Knauss, E., Pelliccione, P., Heldal, R., Ågren, M., Hellman, S., Maniette, D.: 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, p. 43. ACM (2016). https://doi.org/10.1145/2961111.2962639
32. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Softw. Syst. Model. **17**(1), 91–113 (2018). https://doi.org/10.1007/s10270-016-0523-3
33. Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Inf. Softw. Technol. **51**(12), 1631–1645 (2009). https://doi.org/10.1016/j.infsof.2009.04.009

34. Macedo, N., Jorge, T., Cunha, A.: A feature-based classification of model repair approaches. IEEE Trans. Softw. Eng. **43**(7), 615–640 (2016). https://doi.org/10.1109/TSE.2016.2620145
35. Mårtensson, T., Ståhl, D., Bosch, J.: Continuous integration impediments in large-scale industry projects. In: 2017 IEEE International conference on software architecture, pp. 169–178. IEEE (2017). https://doi.org/10.1109/ICSA.2017.11
36. Miller, A.: A Hundred Days of Continuous Integration. In: Agile, pp. 289–293. IEEE (2008)
37. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M.A.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empir. Softw. Eng. **18**(1), 89–116 (2013). https://doi.org/10.1007/s10664-012-9196-x
38. Muram, F.u., Tran, H., Zdun, U.: Systematic review of software behavioral model consistency checking. ACM Computing Surveys (CSUR) **50**(2), 1–39 (2017). https://doi.org/10.1145/3037755
39. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: proceedings of the 22nd international conference on Software engineering, pp. 742–745 (2000). https://doi.org/10.1145/337180.337620
40. Olsson, H.H., Alahyari, H., Bosch, J.: 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, pp. 392–399. IEEE (2012). https://doi.org/10.1109/SEAA.2012.54
41. Olsson, H.H., Bosch, J.: Climbing the "Stairway to Heaven": evolving from agile development to continuous deployment of software. In: Continuous software engineering, pp. 15–27. Springer (2014). https://doi.org/10.1007/978-3-319-11283-1_2
42. Pham, V.C., Radermacher, A., Gerard, S., Li, S.: Bidirectional mapping between architecture model and code for synchronization. In: 2017 IEEE International conference on software architecture (ICSA), pp. 239–242. IEEE (2017). https://doi.org/10.1109/ICSA.2017.41
43. Rapos, E.J., Cordy, J.R.: Examining the co-evolution relationship between Simulink models and their test cases. In: proceedings of the 8th International workshop on modeling in software engineering, pp. 34–40 (2016). https://doi.org/10.1145/2896982.2896983
44. Reineke, J., Stergiou, C., Tripakis, S.: Basic problems in multi-view modeling. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-017-0638-1
45. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: The Epsilon Generation Language. In: European conference on model driven architecture-foundations and applications, pp. 1–16. Springer (2008). https://doi.org/10.1007/978-3-540-69100-6_1
46. Schmidt, D.C.: Model-driven engineering. Computer-IEEE Computer Society- **39**(2), 25 (2006)
47. Seibel, A., Neumann, S., Giese, H.: Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. Softw. Syst. Model. **9**(4), 493–528 (2010). https://doi.org/10.1007/s10270-009-0146-z
48. Stevens, P.: Maintaining consistency in networks of models: bidirectional transformations in the large. Softw. Syst. Model. **19**(1), 39–65 (2020). https://doi.org/10.1007/s10270-019-00736-x
49. Stünkel, P., König, H., Rutle, A., Lamo, Y.: Multi-model evolution through model repair. J. Object Technol. (2021). https://doi.org/10.5381/jot.2021.20.1.a2
50. Tian, F., Liang, P., Babar, M.A.: Relationships between software architecture and source code in practice: an exploratory survey and interview. Inf. Softw. Technol. (2021). https://doi.org/10.1016/j.infsof.2021.106705
51. Torres, W., Van den Brand, M.G., Serebrenik, A.: A systematic literature review of cross-domain model consistency checking by model management tools. Softw. Syst. Model. (2020). https://doi.org/10.1007/s10270-020-00834-1
52. Usman, M., Nadeem, A., Kim, T.h., Cho, E.s.: A survey of consistency checking techniques for UML models. In: 2008 Advanced Software Engineering and Its Applications, pp. 57–62. IEEE (2008). https://doi.org/10.1109/ASEA.2008.40
53. Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R.: Understanding and Facilitating the Co-Evolution of Production and Test Code. In: 2021 IEEE International conference on software analysis, evolution and reengineering (SANER), pp. 272–283. IEEE (2021). https://doi.org/10.1109/SANER50967.2021.00033
54. Zaidman, A., Van Rompaey, B., van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir. Softw. Eng. **16**(3), 325–364 (2011). https://doi.org/10.1007/s10664-010-9143-7

Robbert Jongeling is a PhD student in the industrial software engineering group at the department of Innovation, Design and Engineering at Mälardalen University in Västerås—Sweden. His research interests include industrial applications of continuous model-based development, consistency management and model synchronization. Contact him at robbert.jongeling@mdh.se, or visit https://robbertjongeling.com.

Federico Ciccozzi is an associate professor in Computer Science at Mälardalen University. His research interests cover many aspects of automated software engineering, with focus on model-driven and component-based software engineering for real-time embedded systems. Contact him at federico.ciccozzi@mdh.se, or visit https://www.es.mdh.se/staff/266-Federico_Ciccozzi.

Jan Carlson is a professor in computer science, specializing in software engineering, at Mälardalen University. His current research focuses on component- and model-based development of embedded systems, addressing areas such as optimized allocation, model-level timing analysis and code generation, and the combination of MBD and continuous integration practices. Contact him at jan.carlson@mdh.se, or visit https://www.es.mdh.se/staff/40-Jan_Carlson.

Antonio Cicchetti is an associate professor at Mälardalen University. His research interests target component-based and model-driven software engineering in industrial settings, including model versioning, metamodeling, model transformations and multi-view/distributed development. Contact him at antonio.cicchetti@mdh.se, or visit https://www.es.mdh.se/staff/198-Antonio_Cicchetti.