JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Structural consistency between a system model and its implementation: a design science study in industry

Robbert Jongeling[1], **Johan Fredriksson**[2], **Jan Carlson**[1], **Federico Ciccozzi**[1], **and Antonio Cicchetti**[1]
[1]Mälardalen University, Sweden
[2]Saab AB, Sweden

ABSTRACT During the development of complex systems, several different development artifacts are created and maintained. It is important to gain insight into the degree to which these artifacts are consistent, but this is challenging, especially in complex industrial settings. In this study, we aim to help engineers in their consistency management efforts by creating consistency checks between a system model and the corresponding code. To that end, we performed a design science study in which we develop a consistency checking tool and integrate it into an existing industrial system and software engineering setting. We evaluated the developed tool through a case study in which we measure the consistency before and after its introduction and evaluate the experiences of engineers using it. Our results show that the introduction of lightweight consistency checks into the continuous integration pipeline is beneficial for consistency management in the studied setting. Moreover, we discuss the practical challenges of introducing consistency checks in an industrial setting and find that the majority is of a nontechnical nature.

KEYWORDS Model-based development, Consistency management, Consistency checking, Continuous model-based development.

1. Introduction

During the development of complex systems, several different development artifacts are iteratively created and maintained. We study a setting in which both a system model and its corresponding implementation code are manually defined. Consistency between them is desired; the code shall be consistent with the design prescribed by the system model, and, vice versa, the model shall accurately describe the code so that it can be used as a reference for maintenance tasks.

The literature on consistency checking is extensive and includes approaches focusing on, for example, scalability (Egyed 2001), efficiency (König & Diskin 2017), the ease of defining and maintaining consistency rules (Feldmann et al. 2019), consistency checking with formal guarantees (Leblebici et al. 2017), or heterogeneous types of supported development artifacts (Vierhauser et al. 2012). Consistency checking is indeed a multisided problem and remains challenging to implement in practice. One of the reasons is the complex engineering settings and processes in which the checks should be implemented. This study supplements the existing literature by investigating the challenges of creating consistency checks in an existing complex industrial setting.

More specifically, we report on a design science (Wieringa 2014) study aimed at providing system and software engineers at our industrial partner, Saab AB, with insights into the structural consistency between system models and their corresponding code. We provide a lightweight framework for consistency checks across development artifacts and demonstrate its integration in our industrial setting. In addition, we describe experiences from the implementation of checks and evaluate their effectiveness in improving consistency.

The remainder of this paper is organized as follows. Section 2 includes background on continuous model-based development and motivations for our study. Section 3 details our research methodology. Section 4 describes the first cycle of our design science study, including problem investigation, treatment design,

JOT reference format:
Robbert Jongeling, Johan Fredriksson, Jan Carlson, Federico Ciccozzi, and Antonio Cicchetti. *Structural consistency between a system model and its implementation: a design science study in industry.* Journal of Object Technology. Vol. 21, No. 3, 2022. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2022.21.3.a6

treatment validation, and a first evaluation. Section 5 describes the second cycle, starting with a problem investigation derived from the evaluation of the first cycle, describes in more detail the final consistency checking tool, and includes a case study for its evaluation. Section 6 provides a discussion of the results and experiences of the design science study conducted and reflections on the challenges related to the introduction of consistency checking in an already established complex industrial setting. Section 7 lists related work, focusing in particular on related empirical studies. Section 8 concludes the article.

2. Background and Industrial Setting

The background of this work contains aspects of model-based systems engineering and continuous model-based development.

2.1. Model-based systems engineering in our setting

Model-based systems engineering (MBSE) promotes model-centric design over the traditional document-centric approach. The current *de facto* standard language for systems engineering in industrial applications is the Systems Modeling Language (SysML) (Friedenthal et al. 2014). SysML is used at Saab to create system models that cover both software and hardware concerns.

In our industrial setting, a system model is created to capture the high-level design of the system, its decomposition into separate functionalities, and their allocation to software or hardware blocks. In particular, function blocks consist of one or more system components that are assigned to hardware or software blocks. The behavior of the components is described at a high level, focusing on operations captured in activity diagrams, interfaces captured in proxy ports, and states captured in state machines. Figure 1 provides a simplified overview of the elements that make up a system model. Despite a rather significant conceptual gap between the model and code, the company is interested in checking the consistency between them to facilitate maintenance activities.

In this setting, minimal effort is spent maintaining consistency between the system model and its implementation, and consequently, the risk that they may diverge is fairly high. The way of working is driven by modeling activities, i.e., typically new functionality is added in the system model first and is then implemented in code. Implementing a specific component of the system starts with a handover meeting, in which system engineers and software engineers discuss the part of the system model to be implemented. In these meetings, the supporting material consists of presentation slides on selected fragments of the model. Software engineers typically do not have more interaction with the system model outside handover meetings, and they very rarely investigate the system model due to its size and complexity. After this initial handover, there is no further synchronization between the model and the related code. However, changes occur. For example, software engineers might notice problems with the model while writing code; they typically do not fix the model, but rather adjust their code to ensure a correctly working system. Conversely, changes are made to the system model and shall be propagated to the code, but currently they are not. In both cases, it is easy to lose track of what has changed in both the model and code and to what extent.

2.2. Continuous model-based development

In model-based development (MBD), we include practices in which models are used as core development artifacts. With this definition, we avoid the inclusion of informal artifacts, such as whiteboard sketches, that are used only for informal communication. The crucial aspect is that we expect models to be maintained to accurately describe the system under development and its implementation.

For the development of complex embedded systems, the V-model (INCOSE 2015) with well-gated steps between the subsequent phases is traditionally used. In contrast, modern systems and software engineering practices tend to prefer short development cycles in all different phases of the V-model, with less prominent gates between phases. Typically, multiple engineers collaborate to create the models. In Figure 2 we show the stages of the V-model that we consider in the scope of continuous MBD, selected based on where our industrial partners use models the most.

When adopting short development cycles, one of the most complex challenges is to establish and maintain consistency between various core development artifacts (Jongeling et al. 2019). We distinguish between models (abstractions for a given purpose, following an established syntax with defined semantics) and other development artifacts (such as informal diagrams that do not have an agreed syntax and semantics). MBD provides opportunities to improve consistency management, since it allows more automated transformations between models. However, our industrial partners typically leverage modeling partially and supplement their models with other manually created artifacts, such as natural language requirements, informal architecture diagrams, or spreadsheets. Hence, consistency management often includes time-consuming manual analysis tasks and thereby forms a considerable barrier towards continuous development.

3. Methodology

In this section, we describe the setup of our industry-academia collaboration, the chosen research methodology, and our research questions.

3.1. Research setup

In this study, we cooperate with Saab with the aim of iteratively introducing and evaluating model and code consistency checks in their industrial setting. The first author of this paper has joined (in his role as a PhD student in software engineering) the group of the second author (our main contact person, in his role as a system engineer and software architect) at Saab's premises for approximately one day per week over a period of six months. Before these six months, we collaborated remotely. We had defined the research goal and arranged access for the first author to the company's premises and digital systems. Due to the company's strict security regulations, the access to the digital systems was limited to small example models and a few code repositories. To run the tooling developed on the real

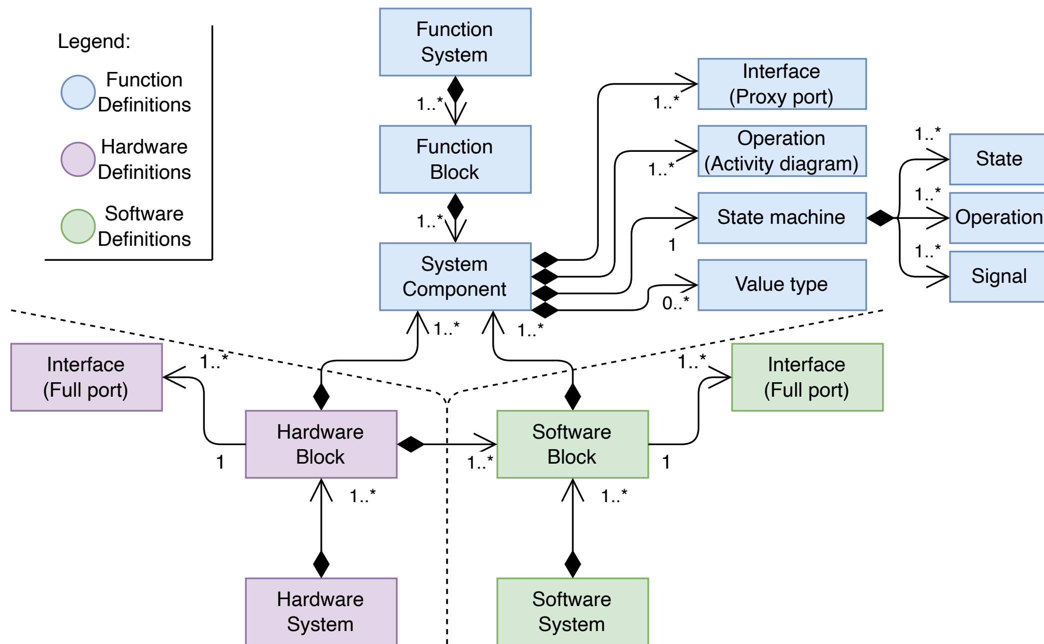

Figure 1 Simplified overview of meta-elements composing a system model. Not depicted are the native, foreign, and test interfaces of function blocks. Based on earlier version of this Figure (Jongeling et al. 2020).

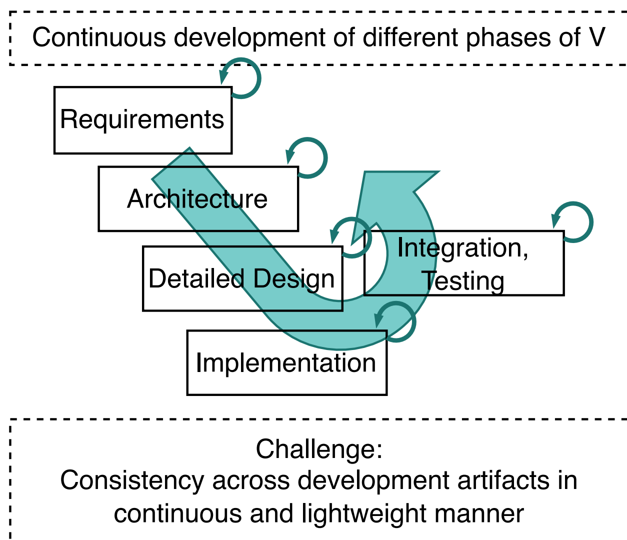

Figure 2 Continuous model-based development entails short development cycles for different phases of the V-model, as well as for the complete V.

system, the engineers at Saab pulled in the latest version of the solution and ran the code on their machines.

To create an approach and evaluate its effect in a real setting, we considered the two research methods of action research (Staron 2020) and design science (Wieringa 2014). Complete integration of the researcher in the industrial setting, as required for action research, could not be achieved and the access of practitioners to participate in the creation of the artifact was limited. Therefore, we chose the design science methodology.

3.1.1. Design Science The purpose of design science (also known as engineering science (Ralph 2021)) as described by Wieringa is to design an artifact and study it in its context. This purpose makes the methodology suitable for studies in industrial settings in which we build knowledge through the engineering of an artifact and its evaluation in the setting in which it is deployed. The advantage of the real setting is that knowledge is built under realistic conditions and unrealistic idealizations (*spherical cows*) are avoided.

Design science research starts by refining a research goal into design problems and knowledge goals. Design problems state how a particular problem context can be improved by designing an artifact that satisfies requirements to help stakeholders achieve their goals. In parallel, knowledge questions are formulated that aim to achieve knowledge goals, which are formulated to help understand the context, the artifacts developed, or their interaction.

The methodology is structured in cycles of four steps. The start of the cycle is an *initial problem investigation* step. In this step, stakeholders and their goals are identified. Moreover, the problem to be solved is defined. The second step of the cycle is *treatment design*, in which the requirements are specified, existing solutions are evaluated, and if no existing solution is deemed suitable, a design for a new solution is created. The third step of the cycle is *treatment validation*, in which the proposed design is validated (note that the validation of the proposal does not require it to be developed). The fourth step of the cycle is *treatment implementation*, which refers to deploying the designed and developed artifact in its context. Thus, this step is not limited to the development of the designed software artifact. In the second iteration of the cycle, instead of an initial problem investigation, now an *implementation evaluation* is

performed. This evaluation can lead to new goals or problems that need to be solved in the cycle in which this evaluation is the first step.

3.1.2. Design Science in this paper In the following, we describe how we followed the guidelines (Wieringa 2014) for design science studies.

In this paper, we describe two engineering cycles, each including "Problem investigation", "Treatment design", "Treatment validation", "Treatment implementation", and "Implementation evaluation". The latter affects the problem investigation phase for the forthcoming cycle. The term "treatment" is used to indicate an approach or tool to address the problem investigated in the studied setting. Our two cycles are illustrated in Figure 3, where we summarize the activities involved in each of the two cycles.

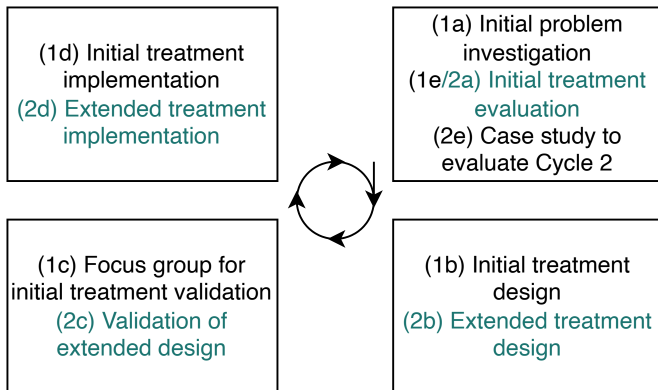

Figure 3 Our steps throughout two engineering cycles (1) and (2), based on the diagrams in *Design Science Methodology* (Wieringa 2014).

3.2. Design problem and knowledge questions

We now discuss our research questions as we formulated them at the beginning of the first engineering cycle. Details on how we deviated from these based on the results of the steps in each cycle are discussed in Section 4 and Section 5.

Our scope was limited from the start to structural consistency checking. Specifically, the objective was to verify that the architectural guidelines as set by the software architects were followed by both the system and software engineers and thereby that the structure of the system model accurately reflected the structure of the code and vice versa. The system model does not specify the intended behavior of the system in enough detail to enable code generation or to check consistency with the behavior specified in the code. Therefore, to achieve consistency, in this study, we aim to maximize the portion of matched elements between the system model and the source code.

As prescribed by the design science methodology, we split our research goal into design problems and knowledge questions. Following the template, we formulate the design problem for our study as: "To

- improve the insight into structural consistency between the system model and code
- by providing consistency checks in the IDE
- that shows model fragments matched to code fragments based on previously agreed architectural rules
- so that system engineers and software engineers can see at which places the system model and code structurally differ."

Our knowledge questions about the designed treatment are an effect question and a requirement satisfaction question:

a. What is the effect on the degree of consistency measured after the introduction of the developed consistency checks?

b. Is our method usable by system and software engineers with minimal impact on their workflows?

We describe the execution of the two engineering cycles in Section 4 and Section 5. An engineering cycle includes the implementation and evaluation of a treatment, and thereby differs from a design science cycle, which includes only problem investigation, treatment design, and treatment validation.

4. Engineering cycle 1

In this section, we describe the stages of problem investigation, treatment design, treatment validation, and treatment implementation of the first engineering cycle of our study. We describe the steps chronologically and outline the aspects of the treatment design that were later changed during the treatment validation.

4.1. Initial problem investigation

In the description of the problem investigation phase, we first describe the background of our industrial setting, including stakeholders and their goals.

4.1.1. Adopting structural consistency checks The team of system and software engineers aims at maintaining consistency between the system model and the code, i.e., the code should conform to the model and, vice versa, the model must conform to what is implemented. The latter is crucial for allowing the model to be a reliable source of documentation so that it can be used for change impact analysis, analyzing the possibilities of reuse of components across different products, and, in general, a reference artifact for maintenance activities.

The current workflow is inadequate to achieve the team's consistency goals due to the lack of tracking changes across the model and code. Therefore, the team could benefit from automated consistency checks between the two artifacts. Our goal is to provide information about any discrepancies between the two artifacts to engineers so that they can be resolved if and when needed. The goal is not to establish complete consistency at all times, since the system model is usually ahead of implementation, and therefore there is generally a need to temporarily tolerate inconsistency (Stevens 2014). Moreover, we establish our approach in an existing setting where we initially expect a large number of inconsistencies, since these were never checked.

The focus in identifying these discrepancies is on the structure of the model and the code. In particular, architects formulate architectural guidelines that can be formulated as consistency rules to give an initial indication of consistency. In this study, the structural consistency between the system model and code is defined as the adherence to the following three rules:

– Each system component should be implemented in a separate repository.
– Each state should be implemented in a separate class.
– Each reception event (signals; these are used in state machines in system components) should be implemented as a case in a switch statement.

In summary, the company is interested in investigating how to adopt consistency checks in their current workflow, if the results provide useful insights and if the results can be used to improve the overall consistency between their development artifacts. Therefore, the two main stakeholders are system engineers and software engineers. Additionally, monitoring consistency over time can provide engineers and managers with data to sharpen otherwise vague refactoring efforts.

4.1.2. Goal definition In this cycle, we formulate the following goal definition: "Improve the understanding of software and system engineers about the consistency between the system model and its implementation by providing structural consistency checks that are frequently executed for engineers to get fast feedback on the consistency between the system model and code, and potentially shorten their development cycles".

4.2. Initial treatment design

We now describe the requirements as elicited by the stakeholders. Furthermore, we discuss several existing treatments, the need for a new treatment, and its design in our setting.

4.2.1. Requirements The main requirement is that our tool should implement three consistency checks corresponding to the rules mentioned above. We further formulate requirements related to the execution of the consistency checks and the visualization of their results. It shall be possible to run consistency checks for a particular part of the system on demand. At the same time, the consistency checks shall be run in the background and the results shall be observable without users running the checks. A final requirement is that the results of the consistency checks shall be accessible to both the software and system engineers using the very same tools that they are already using, thus avoiding adding yet another tool to an already complex development setting.

In our setting, we signal an inconsistency between the model and the code and do not automatically resolve it. One reason is that the model is typically ahead of the code. Therefore, inconsistencies are expected and usually resolved when the implementation catches up. The second reason is that, in principle, in case of inconsistencies, we do not know which of the two artifacts is correctly capturing the intention of the engineers. Our mechanism will thus necessarily fall short of being a bidirectional transformation, since it can not restore consistency due to the partial consistency relationship between model and code, and due to the absence of an authoritative side among the two (Stevens 2020).

4.2.2. Available treatments Below, we discuss several available treatments and why they do not completely match our setting. Therefore, we decided to design a new treatment, which is described in Section 4.2.3.

Despite the required model-code consistency, code generation was not a suitable option in our setting since the system model does not contain enough detail to generate code, and the team does not want to change this. In the past, the team has experimented with generated code in different projects, but never fully adopted this practice due to the low quality of the generated code. The problems observed were with debugging ("*we could run the model on the host, but we could not debug properly on target devices where we can only run the code*") and long-term maintenance ("*projects need to be maintained a long time, up to thirty years, and so we want to make sure the code itself can be maintained so that we do not rely on a modeling tool that may no longer exist*"). Furthermore, there appears to be a need to modify the generated code, which may then result in required updates to the model ("*if you make a change, the round-trip is broken*"). Therefore, the current process involves the manual implementation of parts of the system model in C++ code.

Incremental consistency checking approaches (Egyed et al. 2018) were proposed, among other settings, for model-code consistency (Riedl-Ehrenleitner et al. 2014). Since our main objective was at first to see whether consistency checks are helpful in this setting, we decided not to adopt this approach, since it requires considerable implementation effort. Moreover, we aim to interfere minimally with the existing development setting and notice that the initial challenge of establishing consistency checks in it is related to agreeing on the consistency rules that should hold. Therefore, instead of applying existing approaches or tools, we developed and implemented our own approach, which had the benefit of seeing early results. Anyhow, we keep in mind the theory behind related consistency checking approaches such that in later stages we may extend our approach to benefit from them.

We further consider an available tool that can provide the required consistency checks: IncQuery Suite[1]. This tool provides features to analyze models created using disparate tools in a complex development setting. However, it does not yet seem to provide integration with IBM Rhapsody (as used in the studied setting), and it also does not seem to allow source code to be used among the analyzed artifacts.

Software *reflexion* modeling (Murphy et al. 1995) tools such as JITTAC (Buckley et al. 2013) are capable of building a model of the architecture currently implemented and comparing it with a design provided in real time. A challenge to their adoption in our setting is the need to provide a separate architectural design, in parallel to the already existing system model, on which the reflexion modeling tool can work. Since we want to minimize the overhead for system and software engineers, we opted not to

[1] https://incquery.io/

introduce such an artifact, since it would itself need to be kept consistent with the system model.

4.2.3. Design of new treatment Since no suitable available treatment was found, we propose a new one. Our treatment design can be divided into the consistency checking mechanism and the visualization of the results. The consistency checks will match elements across artifacts, and the visualization will show, for a given code or model element, its matched model or code element, respectively.

A lightweight consistency checking mechanism An overview of the created consistency checking mechanism is shown in Figure 4. We consider the mechanism to be lightweight due to the limited requirements for its implementation and integration in an existing setting. At the same time, we acknowledge the limitations of the mechanism (in Section 6).

The checks are made possible by establishing traceability links between the concrete elements, which is acieved by first indexing the system model and a set of repositories to obtain the model and code elements, respectively. From the codebase, we then index the repositories and the classes and cases within them. From the system model, we index system components, states, and event receptions. Only the elements defined in the consistency rules are indexed; the remainder of the artifacts are not involved. The evaluated consistency rules are those defined in Section 4.1.1. This indexing allows us to check completeness in two directions; we can find elements that are in the model but do not have corresponding code elements and vice versa.

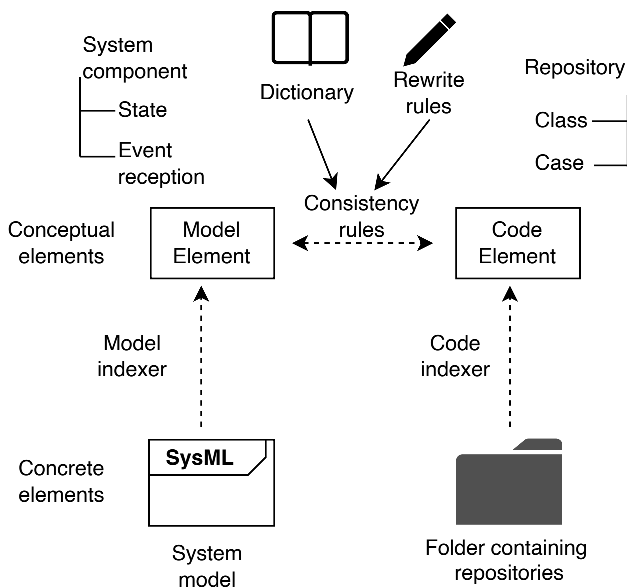

Figure 4 Our approach to structural consistency checking between part of the system model and code.

To evaluate the consistency rules on indexed elements, we rely on name-equivalence. In our implementation, the consistency rules are expressed directly programmed in the consistency checker. We introduce rewriting rules, since the names in the code and the model cannot always be the same due to, for example, white spaces in the names of the model elements. Rewriting is performed on all indexed elements as part of the preparation step before running a consistency check and replaces, e.g., spaces with underscores and removes all casing from names. Moreover, names written out fully in the model are commonly abbreviated in code. Therefore, we also need to provide a dictionary that can extend the abbreviated names of code elements to their full form. In the same preparation step, we introduced this dictionary that, for example, expands the name "hw_mon" in the code into its full form "hardware monitoring" in models. Both these rewrite rules and the dictionary can be seen as a means to accept existing naming misalignments despite the introduced consistency rules.

Conceptual framework for consistency checking We generalize the approach diagram as shown in Figure 4 and show in Figure 5 our proposed lightweight consistency checking framework between various development artifacts. The figure shows a framework for consistency checks between concrete development artifacts, such as a "system component" in the SysML model and a "repository" in the codebase. The framework can be instantiated in the same way to check the consistency between, e.g., states in the model and classes in the code. Figure 4 is such an instantiation, where we show how the consistency rules are implemented for particular concrete development artifacts.

The bottom layer of Figure 5 denotes concrete elements that can be part of development artifacts. In our setting, the concrete elements are those of the system model and the codebase. Reasoning about consistency between elements is done in terms of conceptual elements, rather than check between a specific model element and a specific code element. Indeed, the desired consistency is expressed in terms of, for example, "each system component shall be implemented in a separate repository", rather than "system component x shall be implemented in repository y."

The indexers represent a mapping from a set of concrete elements to conceptual elements. In the example, the model indexer retrieves concrete elements from the system model that are considered system components. To do so, we require a definition of what a system component is exactly; in our setting, it is defined as a block with a stereotype "system component". Similar definitions are required to mine the other elements and store them in the element structure, as shown in the upper level of Figure 5. This structure allows storing elements in a hierarchy. Storing the parent of elements is needed to match, e.g., a state to a class inside the repository that is matched to a system component and not to a class in a different repository. We cannot assume that the names of concrete elements are unique. Therefore, during mining, for each state and event reception, the parent system component is kept as the parent model element.

The proposed framework is instantiated here to illustrate a consistency check between two elements, but can also be extended to include more elements from other development artifacts. In future work, we intend to extend the current implementation to include other artifacts that the team develops and wants to keep consistent with the system model, such as XML files containing interface definitions and VHDL code.

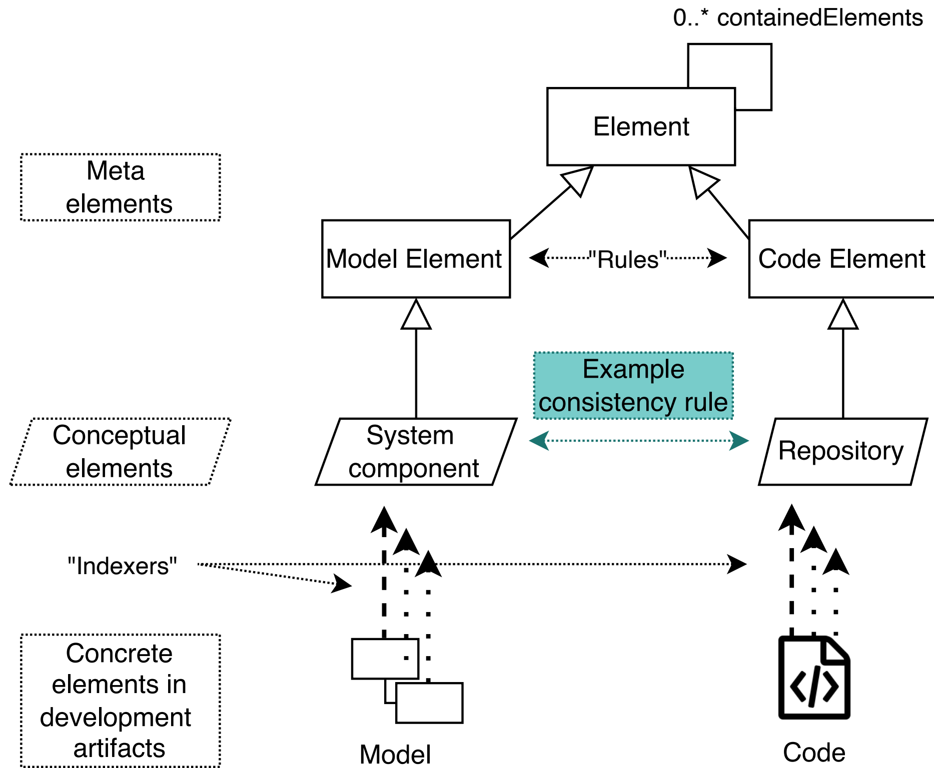

Figure 5 Conceptual framework for light-weight consistency checking of diverse development artifacts.

In the framework, we can regard the conceptual elements as a metamodel of the concrete elements that conform to them. We can further generalize the conceptual elements to denote that they are model or code elements, which are again generic elements. This structure shows how we can benefit from modeling techniques for the creation of consistency checks across diverse artifacts. One of the reasons why we consider this framework to be lightweight is that it does not require formal or complete transformations between artifacts. Rather, the indexers are used to cherrypick from the concrete elements exactly those of interest for consistency checking. Thereby, the consistency rules can deal with development artifacts that are not necessarily well-behaved models in the MDE sense. Instead, since the indexers are not attempting a complete or formal transformation of the concrete elements, they allow for the development artifacts to be incomplete, informal, or even not well-formed. Dealing with such artifacts is crucial in continuous MBD settings, since all development artifacts are continuously evolving and demanding their correctness at all times would be too onerous for the development process.

Results visualization We provide only a high-level description of the initial visualization design since it was changed after the first validation of the treatment, as discussed in Section 4.3. Our initial visualization design showed the results of consistency checks inside the IDE for software engineers and inside the modeling tool for system engineers. To achieve this, plug-ins for CLion and IBM Rhapsody were planned that, when selecting a code or model element, would show its related model or code element, respectively. For example, when opening a C++ class, the plug-in would include a graphical view of the related portion of the state machine in the system model. Software and system engineers would then interact with the consistency checks from their respective tools that they already use for the majority of their tasks.

4.3. Focus group for initial treatment validation

Validation here refers to an upfront investigation (i.e., before implementation) of the suitability of the designed treatment for the problem it aims to solve. Evaluation of the implemented and deployed artifact is performed at a later stage of the engineering cycle. We have validated the proposed treatment by expert opinion. Due to the considerable effort required to implement even a minimal prototype, it was deemed better to invite a panel of representative users to share their opinion on the initial treatment design and then adjust when needed.

To validate the treatment design, we organized a one-hour session with four engineers from the company and one researcher (the first author). In addition to our main contact, the other three engineers (two software engineers and one system engineer) were new to our study and the proposed design. The researcher first introduced the background and motivation of the project and then the treatment as described earlier in this section. A round of questions from the engineers ensured that the presented design was clear to all participants. Lastly, engineers were asked to imagine the solution being implemented and to answer the validation questions mentioned above. The company

has strict security regulations and does not allow audio/video recording on site. Hence, the researcher took extensive notes during the meeting and worked them out immediately afterwards. We now discuss the results of the meeting based on four types of knowledge questions used to validate the treatment.

4.3.1. Effect questions

The purpose of *effect questions* is to investigate the effect of an artifact once it has been deployed in its context, i.e., is it useful? Since the artifact has not yet been deployed, this validation relies on the predictions of the engineers based on their mental model of the design. The engineers foresee several benefits of introducing the designed treatment in their current project for preventing process and architecture erosion, and for an improved insight into model-and-code consistency. We elaborate in the following.

The first benefit foreseen is to counteract process erosion. Engineers indicated that visualizing inconsistencies can prevent deviation between model and code caused by not following the development process or the preagreed architectural guidelines. "*When we started this project in 2015, there was a better feedback loop between system and software engineers. This tool could enhance or reinstate that feedback loop again, by indicating inconsistencies over time, as the model and code are updated.*" One system engineer also mentioned the long time interval between model and code updates and the difficulty in keeping track of changes. "*Now, things are improved in the code and then forgotten about. Then six months later, you see that the model is not at all updated. So some reminder is needed to show that the model is not correct anymore.*" Indeed, consistency checks were deemed helpful, even for parts of the system. "*To just have these structural checks for even half the system components would be beneficial.*" Another area of interest was the connection between the code and the model. This was particularly of interest to software engineers, who noted that they currently have limited knowledge of the system model. "*This will help me to learn to know the model and understand the model better. Currently, I do not look at the model.*" Another software engineer said: "*This will help me to see what has changed in the model and will give some help to see how complete the implementation is.*"

4.3.2. Trade-off questions

The purpose of trade-off questions is to determine the effect that alternatives to the designed artifact would have in the context; what is the difference between them? Since we did not find suitable treatments, we can only compare with the current state of practice, which involves manual checks, not driven by the development process or supported by tooling. Essentially, after the handover between the system and software engineers, the consistency between the model and code is no longer checked. In some cases, software engineers provide feedback to system engineers, e.g., to mention a mistake in the system model, but the model is then usually not updated to reflect the code change; this introduces inconsistencies.

4.3.3. Requirements satisfaction questions

The purpose of requirements satisfaction questions is to determine whether the effects produced by the artifact actually satisfy the requirements. The discussion of the satisfaction of functional requirements as we listed before was short; the engineers agreed that the design meets the requirements. We then had a longer discussion about non-functional requirements that were not formulated before, related to the maintenance and the scope of consistency checks.

The main objection foreseen was related to the maintenance of the consistency checks themselves. "*As a system engineer, I would be unhappy with maintaining rules if it means to change a term in multiple places.*" Indeed, we should ensure that the definition of consistency rules does not create a burden on system and software engineers. Related to the maintenance of the artifacts, the engineers identified the need for a filtering of the results to prevent an overwhelming list of inconsistencies. Moreover, it was identified that starting with a tool earlier would result in fewer inconsistencies. "*If started from the beginning, there wouldn't be so many errors. But now there will be many and many errors.*" Similarly, it was discussed that some filters are needed to select the elements that need to be checked (e.g., everything within this particular package).

The engineers furthermore expressed a desire to check only part of the model. "*It is important to be able to select what part of the model and what part of the code you want to check.*" At the same time, there was a desire to extend the scope of consistency checks beyond the system model and the C++ code. "*Not all implementation is in C++, there are also parts of the system implemented in e.g. VHDL.*" The need to eventually support more types of artifacts than only the system model and code is one of the reasons why we eventually set out on a slightly different path for the implementation, as we shall see later.

Finally, some new opportunities also emerged from the discussion. In particular, engineers noted that the overview of the consistency checks could also be very useful for communication. "*Some percentage, summarizing number, or KPI for consistency would be very useful for tracking the progress over time and to be able to communicate in a simple way to management the improvements (and to make a case for a need to refactor).*"

4.3.4. Sensitivity questions

The purpose of sensitivity questions is to determine how the artifact would respond to changes to the context, such as larger or different projects.

We discussed the applicability of the proposed tool to other projects in the company. One concern was related to the models involved, which are not everywhere of the same depth and structure: "*In other projects were I worked we did not model in such a formal way. If you just have some* Visio *diagrams, you cannot check much.*" However, there is a project in the company that has a much larger scale than the project we are working with in this study, and for that the engineers saw no objections to using the same approach, provided the aforementioned means to filter the results. If we further change the context, for example, by changing the MBSE way of working or the programming language, then the application of the proposed treatment will be more difficult.

4.3.5. Conclusions from the meeting

We did a validation workshop for the proposed bridge between CLion and Rhapsody. The result was a change in the prioritization of features for two

main reasons: (i) consistency across more artifacts should be checked, and (ii) a complete overview of all inconsistencies and some statistics should be prioritized over the inclusion of a visualization of system model parts in the IDE. Hence, we have changed the scope considerably and, in particular, limited the requirements that we will include in our tool to the minimum required for evaluating the effect of consistency checks. Therefore, we adjusted our goal and aim at first indexing model and code elements, then running the consistency checks according to three predefined rules by the stakeholders, and eventually to present the results in a single overview page.

4.4. Initial treatment implementation

Implementation here does not refer to developing the proposed tool, but rather to applying it in our industrial setting. Nevertheless, we first briefly describe how the tool was developed.

4.4.1. Developed tool The tool follows the approach shown in Figure 4, which is an instantiation of the generic framework shown in Figure 5. In particular, we have implemented a model indexer that retrieves system components, states, and event receptions from the system model, and a code indexer that retrieves repositories, classes, and case statements from the code base. The matching of elements follows the maintained hierarchy. In our case, a class C is only matched to a state S if the repository that contains C is matched to the system component that contains S. This hierarchy allows distinguishing between components with the same name in different parts of the system, and thereby facilitates their correct matching.

Consistency checking The model indexer uses an API provided by Rhapsody that allows opening a model in a headless instance and performing a depth-first traversal of all model elements. For each type of model element for which we have defined a consistency rule (i.e., “system component”, “state” and “event reception”), we create a “model element” object. During mining, we keep track of the hierarchy of model elements, i.e., which system component is the parent of the mined states and event receptions.

The code miner recursively traverses all folders in a provided root folder. Folders containing “.git” are considered repositories, and within repositories, system components are identified by those folders that contain definitions of states and attributes header files. When a folder is deemed to be a repository containing the implementation of a system component, a “code element object” is created for it. Then, within that repository, we traverse all code and header files, line by line, to find class declarations and case statements (in switch statements). For each of these classes or cases, a “code element” object is created and the hierarchy is maintained too, meaning that we keep track of the repository that this class or case was found in.

The consistency checks, similar to the indexers, are implemented in Java. We use the user-provided dictionary and the rewriting rules as discussed in Section 4.2.3 before to match elements. First, we try to match the top-level elements, i.e., “system component” model elements, and “repository” code elements. Once the top-level elements are matched, we try to match the elements contained within them. As a side note, for

+ H:\git\project.component
Contained Classes
+ H:\git\project.component\public_include\componentstates.h\aMatchedState
+ H:\git\project.component\public_include\componentstates.h\anUnmatchedState
+ H:\git\project.component\public_include\componentstates.h\aClassAddedInCode
Contained Events
+ H:\git\project.component\src\componentstates.cpp\anUnmatchedEvent
+ H:\git\project.component\src\componentstates.cpp\aMatchedEvent

Figure 6 Anonymized sample of consistency check results, showing code elements and if they are (green) or are not (red) matched to a model element. The top-level element is colored yellow to indicate its contained elements are partially matched.

testing and quick rerunning of the consistency checks, it is advisable to store the indexed model and code elements such that the indexing task does not have to be repeated. In our case, we persist a model index and a code index each time we mine the model or code, which proved to be very useful during debugging and for creating new visualizations of the results.

Results visualization An HTML page is generated with a visualization of the results; an example is shown in Figure 6. We show the top-level elements (system components and repositories) and allow us to expand and collapse them to show and hide the elements contained in them. The contained elements are marked in green (for matched code/model elements) or red (not matched). The top-level elements are shaded between green and red, indicating the range from a higher to lower degree of consistency, based on the number of matched containing elements. When top-level elements are not matched, they are red since none of the containing elements will be matched. The page also contains a list of system components and their matched repositories, and vice versa, for easy lookup. Lastly, the page shows statistics on the number of mined elements of each type and the percentage of matched and non-matched elements.

Roll-out We developed our tool and deployed it in the project in our industrial setting. For the initial deployment, we executed the tool only on the machine of one system engineer, to gather a first feedback.

4.4.2. First feedback Once we created the first consistency check results, there were immediately some useful insights to be gathered. The engineers could see some parts of the model and code that were inconsistent, even though they were expected to be consistent. The results page gave them a way to explore the model and code from a different perspective.

We found that the rules provided by the domain experts only matched a tiny fraction of indexed elements (in the order of 30 out of 1000). There are two main causes for this discrepancy; the first is the incomplete dictionary, which accounts for a portion of missed matches, and the second is that the architecture erodes and does not follow the rules exactly. Throughout the system evolution, engineers have made many decisions on the structure of the model and code; some of them follow the architectural guidelines, some may be acceptable deviations, and

some inconsistencies. Consequently, it is now challenging to capture these decisions appropriately in a set of consistency rules. In some cases, matches that should be made even without dictionary entries were not made because of this lack of adherence to the architectural guidelines. The results confirm the initial expectations of the engineers about a large number of inconsistencies. With many errors to be displayed, filtering is required to properly navigate the results. To support filtering, we display the results as collapsible rows in a table so that the overall size of the collapsed list is smaller and more easily navigable.

4.4.3. Changes to the treatment made during implementation We made some changes during the implementation of our proposed treatment, in particular related to the indexing of concrete elements. More specifically, we encountered an unexpected challenge in creating the model and code indexers due to the need for several iterations to define clearly how the conceptual elements (system component and repository) map to concrete development artifacts. For example, according to domain experts, the top-level code components should be Git repositories, and hence each system component in the model (a block with stereotype "system component") should be matched to a Git repository. Soon, exceptions to this rule appeared, because in some cases multiple system components were included in a single repository. Then we considered whether these inconsistencies are due to an incorrect rule or an incorrect code structure. In this case, the domain experts agreed that the rule should be refined and, hence, the miner was changed to consider a different heuristic for mining the top-level code elements. We changed it to identify as top-level code elements all folders that are in a Git repository and themselves contain a "public_include" folder that contains <x>states.h and <x>attributes.h files, where x is the name of the system component. It was interesting that this mapping was not at all easy to determine and took several iterations for the model elements, too.

4.5. Initial treatment evaluation

As seen in Figure 3, the evaluation at the end of the first cycle is at the same time the first step and problem investigation of the second cycle. Now, we describe only the implementation evaluation. This evaluation focuses on the treatment after implementation in our industrial setting.

Since we have iteratively developed the approach and tested it with the help of system engineers, we did not do a separate evaluation at the end of the implementation phase. Throughout development, we have improved the number of mined elements by improving the definition of conceptual elements. We also included increasingly more entries in the dictionary, leading to more matched elements. The researcher and system engineer then performed a manual evaluation, sampling some of the results, and looking at the corresponding code. In all cases, inconsistencies were correctly identified.

The results were considered very useful: "*Without this, we wouldn't even know that there was an inconsistency. Having an aid, especially a visual aid, really helps us verify structural consistency, ensuring that things are named correctly.*" It is exactly that kind of structural inconsistency that we aimed to identify with our approach. This iterative development-and-evaluation approach has led to new desired features, which we will discuss at the beginning of the second engineering cycle, as elaborated in Section 5.

5. Engineering cycle 2: continuous checking

In the second cycle, we focus on the necessary improvements to the consistency checks as emerged from the evaluation of the first cycle.

5.1. Problem investigation (coincides with the initial treatment evaluation)

The evaluation of the first cycle showed that the results of the consistency checks could be useful. Therefore, we continued our work and in the second cycle addressed some problems that were identified too. In particular, we focus on the accessibility of the consistency checks and their results to software and system engineers without being interrupted in their development tasks. To prevent consistency checks from being neglected or their results not being visible to stakeholders, a new goal was formulated on top of the one from Cycle 1, as follows.

Goal definition: Improve the availability of the consistency check results, such that all software and system engineers of the team can, at any time, explore the latest state of consistency between system model and implementation, without having to run the tool locally.

5.2. Extended treatment design

To improve on locally run consistency checks as created so far, we proposed to include their execution in the Continuous Integration (CI) pipeline. For the development of the code, the team uses the Git-based tool Bitbucket and Jenkins to automatically run builds and tests. To minimally intrude, we propose to create a Jenkins job that calls our miner and runs the consistency checks after each build. We implement the consistency check as a post-build step to ensure that we do not slow down any of the other automated checks. This also ensures that the execution time of the checks is not particularly important. The miners can take quite some time if the model or other artifact is very large. We do not fail the build based on the results of the consistency check, since it is intended as an overview to be provided to the engineers.

5.3. Validation of extended design

In this cycle, the treatment validation was less extensive than in the first one, mostly because we already had gathered the needed input then. Therefore, the validation in Cycle 2 consisted only of informal discussions with the engineers. No changes to the designed treatment were made based on this validation.

5.4. Extended treatment implementation

Due to difficulties in setting up Jenkins with correct permissions and access to the required tools, we could not yet include consistency checks in the CI pipeline. Since the most important aspect

was to provide access to the results to all engineers, we simulated instead this effect by having our main contact share the consistency check results once a day with the other engineers involved in the case study.

5.5. Case study to evaluate Cycle 2

For the reasons listed in Section 4.2.2, adopting existing treatments in the studied setting was not possible. Hence, a comparative evaluation with existing treatments was impractical too. Therefore, we limited our evaluation to a case study in which we observed the use of our tool, as implemented in Cycle 2, in the real setting. This evaluation was more extensive than the evaluation after Cycle 1 and it included both a quantitative and qualitative evaluation of the tool. The quantitative evaluation focused on measuring the effect of consistency checking when letting an engineer work on refactoring tasks using our tool. The qualitative evaluation focused on studying the usefulness of the results and the usability of the tool for engineers in their current setting.

5.5.1. Quantitative evaluation To measure the effect of using the consistency checking tool, we first measured consistency in the past, i.e., we executed our consistency checking tool for checked-out past revisions of the model and code. We measured the consistency every week for six weeks prior to the introduction of the tool to observe what happens to the consistency if no particular attention is paid to it and no tool is available to show inconsistencies. Thereafter, to see the effect after the introduction of our tool, we asked engineers to work with the tool and record the time they spent on refactoring model or code. Subsequently, we again measured the consistency to observe the effects.

Figure 7 shows the percentage of correctly matched model and code elements in six weeks prior to the introduction of our tool, and the improvement after its introduction. The percentage of matched code elements (dashed lines) is higher than the percentage of matched model elements (solid lines) because the total number of matched elements is greater (in the last week, 48 top-level code elements and 219 system components are indexed). Throughout the past weeks, we see a fairly stable degree of inconsistency between the model and code. There are minor changes after smaller iterations, but the only significant change is seen after the introduction of our tool in week 7.

The engineers put in cumulatively 2 hours of work using the tool and working on resolving inconsistencies. The result was an increase in the number of mapped system components from 15 to 22 (out of 219 components), the number of mapped states from 40 to 49, and the number of mapped event receptions from 42 to 62. In summary, the effect is an increase in consistency between the model and the code from 7% to 10% after 2 hours of work with our tool. The numbers shown correspond to the number of indexed model elements and the number of matched code elements. We also measured in the other direction and observed the same improvements in the last week. As expected, overall consistency is low, due to a lack of feedback between the developers working on either artifact.

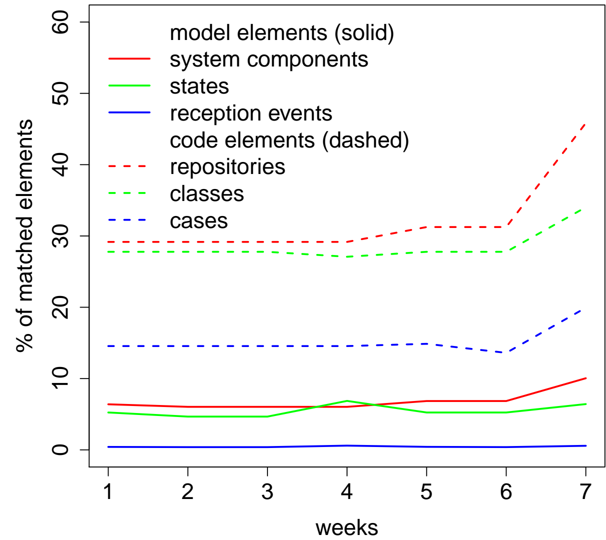

Figure 7 Consistency as expressed in the percentage of matched elements in last 6 weeks before introducing the tool, and in week 7, after introducing the tool.

5.5.2. Qualitative evaluation In addition to the measured effect of introducing consistency checks, we evaluated the developed tool in a one-hour focus group session including the researcher and four software and system engineers. During the meeting, the researcher first introduced the purpose of the meeting and the developed tool. One of the engineers, who worked with our tool before, demonstrated its use to the other engineers. We then discussed with the engineers their thoughts on four aspects: (i) the usefulness of the consistency checks themselves, (ii) the usefulness of the integration of the checks in the CI pipeline, (iii) the usefulness of the presentation of the results (consistency check outcomes), and (iv) the most useful next steps.

Usefulness of results During the introduction and demo, a discussion about the desired consistency between model and code was already started. An engineer remarked: "*this puts a requirement on following a specific naming convention. Not everybody does this.*" We found a similar result that was obtained earlier in the study; overall, it is a challenge to find an agreement among engineers on how precise consistency checks should be and to what extent architectural guidelines and naming conventions should be enforced. One of the identified uses of the tool was to serve as an incentive for improving consistency.

Nevertheless, the engineers agreed that if model-and-code consistency is desired, then the provided consistency checks are effective. "*A tool like this is absolutely needed. This refactoring work can not be done manually. And this implementation seems to be the best first step towards it.*" The checks were found particularly useful in identifying specific locations in large models and code bases that needed maintenance. "*The tool indicates where to look, something is up over there, I should investigate.*"

The use of these consistency insights could also be used to track implementation progress. "*We could use these checks to see how far we are from implementing the model. To see if we are done or very far from it.*" Moreover, the engineers saw practical value in consistency checks for refactoring tasks. "*We have an additional use case for this tool in the next sprint when we are going to refactor a component (in the code) to better match its design in the model.*"

Usefulness of integration in CI pipeline The engineers found the integration in the CI pipeline valuable, thanks to its enabling for a longer term monitoring of consistency. The web page with results is available to all engineers and does not require any specific tool to view. The engineers indeed appreciated this accessibility and proposed several ideas for visualizing more aspects of the results. For example, it was proposed to include a colored box stating the consistency percentage as part of the build results, perhaps separated for model-to-code and code-to-model consistency.

Unexpectedly, the engineers even discussed the possibility of failing the build, upon newly identified inconsistencies. "*If you want to enforce consistency you can fail the build step if anything is inconsistent.*" Eventually, the engineers agreed that there are several scenarios in which inconsistency must be tolerated, and hence strict gating on the checks was not sought. We did not even expect the discussion because of the commonly agreed view in the literature that inconsistencies must be tolerated so as to not interfere with development (Stevens 2014).

Usefulness of results presentation The presentation of the results was not prioritized in the tool development. The engineers proposed several ideas to improve the visualization of the results. Among them, there were suggestions for showing, in the build results, colored boxes with consistency numbers for different teams and for function blocks. Another suggestion was to return to our initial idea and include the links to model elements within CLion. Software engineers also expressed their challenges with the system modeling tool and their preference not to have to deal with it.

Useful next steps During the evaluation sessions, the software and system engineers appreciated the consistency checking framework to illustrate the implementation ("*simple is good*"), particularly because they were worried about maintainability of such a tool. "*It remains to be seen if the overhead of maintaining this is worth the results.*" None of the engineers have worked on the tool yet, but several expressed interest in further developing it. In particular, we are discussing to adapt the tool to be used in a different and notably larger project. The need for adaptation comes from the fact that the targeted project uses different architectural guidelines for implementing system components and state machines. Hence, there are different consistency rules, but also different indexers required, since different top-level code elements need to be considered.

6. Discussion

The consistency checking approach presented is lightweight. Minimal interference with the existing industrial setting is needed. In our setting, the limitations brought about by this lightweightness are currently acceptable, since the aim was to provide engineers with consistency information and to understand whether it would help them improve consistency. During our study, we collected several take-away messages for engineers and other stakeholders that aim to address consistency checking in an existing setting; the remainder of this section discusses them.

6.1. Take-away from developing consistency checks in an existing setting

The process of developing consistency checks in the company has been helpful to software and system engineers for reasoning about the structure of the project and the desired architectural consistency. In particular, the conceptual framework (Figure 5) provided a good starting point for discussions about the definition of indexers. Indeed, defining mappings from conceptual to concrete elements to index them turned out to be one of the main challenges of implementing the approach in the existing setting. It took several iterations to arrive at suitable definitions of the concrete elements denoting "system components" in the model and their counterparts in the code. A suitable definition requires us to capture the intended structure of the model and code, but at the same time it should handle the existing way things are implemented if those ways are acceptable too. It is challenging to get all stakeholders to agree on what the scope of concepts such as "system component" is exactly, as it is a domain-specific and setting-dependent concept; there is no concept of a system component in SysML.

Similarly, there is a challenge of agreeing on the consistency rules themselves. For example, in our setting, stakeholders may not agree that each system component must be implemented in a separate repository. Indeed, also here, the challenge is rooted in the collaborative nature of engineering large systems. In our setting, we followed the architectural guidelines that the engineers had designed at the beginning of their project. However, during the implementation and evaluation of consistency checks, the list of exceptions to the rule continued to grow. Then, there is a need for expert engineers to decide if something is an inconsistency or if it should be allowed as an exception to the rules.

An interesting observation from the early implementation is that there is a trade-off to be made between how good consistency checks should be at identifying correspondences between model and code elements versus the usefulness of the outcomes, i.e., consistency checks should not be *too* good at matching elements. Indeed, if consistency checks are too lenient and identify consistency between elements that are named only approximately the same, it does not help the human understanding of the artifacts. In the extreme case, consistency checks could judge specific model elements and code elements as consistent for reasons that a human engineer does not understand. Then, the checks would defeat their purpose.

Our approach differs from concepts in which consistency is

aimed at being restored (Stevens 2020). In our setting, consistency checks provide a quality metric for a set of interrelated development artifacts. In contrast to other views, we do not consider consistency restoration as a driving force in the engineering phase.

6.2. Framework implementation choices

Once the relationships between concrete and conceptual elements are clear and the consistency rules to be checked are clear, the implementation of the framework can be done in many ways. For expressing and evaluating the consistency rules, even existing constraint languages can be used, provided that the conceptual elements are expressed in forms that those constraint languages can reason over. In this study, we used a general-purpose language (Java) to implement the miners and consistency checks. The main reason for this choice is that we used the Rhapsody API to mine the model elements from the system model. To visualize the results, different approaches can be taken.

Including consistency checks in the CI pipeline is expected to work well in this setting, given the feedback of the engineers on our work-around scenario. Real-time feedback is not really needed for system components in the model that will be implemented much later. Moreover, we are deploying this approach in an existing industrial setting. The engineers have indicated their expectation that, initially, a large number of inconsistencies will be identified because consistency has not been prioritized thus far. In that sense, engineers prefer to have an overview to be checked on demand (pull) rather than being continuously notified of all inconsistencies (push). Moreover, in our evaluation, we showed that introducing even a few simple checks already provides engineers with new insights about the relations between the system model and the code.

Our main goal was to provide consistency checks to assess whether it would be beneficial in this setting. Therefore, during the programming of the proposed approach, we made several design decisions aimed at keeping the development time low, in favor of having early results to guide the way forward. Limitations of the tool include the lack of incrementality of the consistency checks; i.e., each time they are run, both the model and code need to be indexed from scratch. In Egyed's work (Egyed 2010), instant consistency checking is achieved by instantiating consistency rules for each pair of elements that should be consistent and then reevaluating only those rule instances that are affected by a change. Even if we did not focus on incrementality, we could utilize the hierarchy of conceptual elements to determine their scope. For example, if a state is changed, we only need to re-index the elements in its containing system component. In this sense, the miners can be made incremental, provided that we can access the changes to an artifact compared to its previous version and maintain an index of mined elements for each artifact.

6.3. Experiences of performing a design science study

We were studying a real industrial setting and soon realized the need for the development of a new solution. There is a danger of such endeavors becoming very informal, which threatens the validity of the created knowledge. Therefore, our choice for design science was mainly motivated by the wish for a structured research method involving the engineering of a software solution and its evaluation in a real setting.

The cyclic nature of the methodology has the advantage of producing some results in early phases, which can be good, especially in industrial collaborations where researchers may need to motivate their use of company resources. Moreover, the methodology allows us to create a solution that fits in the studied industrial setting while producing knowledge that can be applicable beyond the specific setting. For us, the value of the methodology followed lies in the ability to study the problem in a real context, thereby supplementing the academic literature that often necessarily relies on constructed examples.

In our experience, any industry-academia collaboration requires a significant start-up period in which the problem to be studied is defined. If researchers are external, this start-up time may be extended due to potentially time-consuming activities such as getting access to the company premises and digital systems. During the collaboration, further challenges can be encountered due to confidentiality, researchers working at company premises, licensing and ownership of developed products, and changing of preagreed time commitments from either side. Later, conflicts between the required confidentiality of the industrial setting and the required openness of research may pose challenges during the dissemination of the work. Unfortunately, there is no silver bullet to avoid these challenges, but of crucial importance is timely and frequent communication.

6.4. Threats to validity

Unfortunately, constraints by the industrial partner limits us in sharing the developed tool. Moreover, we were not allowed to record the interviews, and therefore the quotes are based on extensive notes by the first author. We now discuss *internal* validity (are the effects really caused by our treatment or are there other factors in play?) and *external* validity (can our treatment and results be generalized to other settings) of our study (Feldt & Magazinius 2010).

Internal validity To ensure internal validity, we want to ensure that it is our approach and not other external factors or random noise that leads to improved consistency. To do so, we compared the consistency between the model and the code before and after the introduction of our tool. Throughout the revision history, we observed a steady or gradual decrease in inconsistency. Indeed, consistency seems to follow Lehman's seventh law of software evolution (Lehman 1996), and only deteriorates when nothing is done to counteract it.

External validity A consequence of the chosen research methodology is that the generated knowledge is not universally generalizable, but falls within the "middle range" (Wieringa 2014). In the best case, generalization is limited to settings similar to that studied in this article. We believe that our approach can indeed be generalized to other settings, since it can be applied to check the consistency between any type of artifacts. The specific instance shown in the study is hard to apply to other settings, since it is based on consistency rules for concrete

elements in the specific setting. However, the MBSE setting at the company is based on common standards, so there are likely similar settings that our approach can be adopted in.

The validity of the result is further dependent on the qualitative and quantitative evaluation. We attempt to prevent the threat of taking a snapshot of the consistency state by measuring consistency in the weeks leading up to the introduction of the tool, in which no particular attention was paid to consistency management. The engineers indicated that consistency management was virtually impossible without tool support, but there is still a risk that during these weeks improvements would have been found if they had been trying their best to resolve inconsistencies. The improvement in week 7 indicates that the tool can result in a higher proportion of matched elements, which is what we wanted to evaluate. We expect further improvements beyond week 7, when the developers continue to use our tool, are more aware of inconsistencies, and slowly start to resolve them whenever they are making changes to related elements of the code or the model.

7. Related Work

In addition to the discussion of available treatments in Section 4.2.2, here we discuss other related works. Since our focus in this paper is specifically on the introduction of consistency checks in an existing industrial setting, we avoid an elaborate discussion of the related work on consistency checking approaches. Instead, we refer to recent secondary studies in this and related domains, e.g., (Cicchetti et al. 2019; Muram et al. 2017; Torres et al. 2020). Moreover, in the remainder of this section, we focus on other studies involving practical evaluations on consistency checking approaches in industry.

A study in which consistency checks are introduced in an industrial setting found that one of the main challenges is to interact with domain-specific and proprietary tools in an existing setting (Demuth et al. 2016). We see similar challenges in our setting, but foresee that our generic framework requires minimal features in any existing tool. What we need in all cases is to mine the information either from the tool in which an artifact is created or from its persisted form. Since we in particular want to support artifacts that are not necessarily well-formed models, we consider mining information as a minimal step to enable a consistency checking mechanism.

A similar study as ours has looked at improving the way of creating and maintaining consistency checks in an MBSE setting by using a linking language based on triple graph grammars (Feldmann et al. 2019). Their findings indicate that their approach is more usable than writing textual rules (in EVL). In our study, we discuss the creation of the rules as an architectural challenge and do not consider that each engineer will create and maintain links individually.

The questions we are posing in a real industrial setting have recently also been studied in a controlled experiment, where inconsistency feedback was shown to help developers in co-evolution tasks (Kanakis et al. 2019). While our study also considers model-code consistency, the difference is that in our case, the abstraction gap between model and code is large and, therefore, in the normal way of working, there is a longer time between the change in the model and the corresponding change in the code. This is also the reason that many of the related UML-to-code consistency checking work (Usman et al. 2008; Lucas et al. 2009) is not directly applicable in our setting, the code is expressively not generated from the model in our setting. If it could, then we would not need to perform these consistency checks.

A study on architectural consistency among practitioners identified as one of the challenges the reluctance to accept formal consistency checking approaches due to a perceived large effort in adopting them (Ali et al. 2018). Our approach can help practitioners set the first steps towards consistency checking by using a lightweight framework. If and when it is proven useful, practitioners can then opt to use more formal approaches and more extensive tools.

8. Conclusion

In this section, we summarize the main learning outcomes and discuss future work based on the findings of this study.

8.1. Main learning outcomes

In this paper, we report on two design science cycles in which we introduce structural consistency checks between a SysML system model and its corresponding C++ code base in an industrial setting. We have addressed the design problem by providing software and systems engineers with insight into the consistency between a system model and its implementation. Through the evaluation of our implemented artifact, we have seen that it has a positive effect on the quality of the artifacts (i.e., the number of identified inconsistencies decreased). Moreover, we have designed the artifact in a way that minimally disrupts the current way of working of engineers when adopted in their existing setting. Our implementation is an instantiation of a more general framework for consistency checking between diverse development artifacts, which we also present in this paper.

During the collaboration with the industrial partner, we found that a great challenge is to define the mapping from conceptual to concrete elements. Although it was quite clear what the consistency rules should be at the conceptual level, it took several iterations to formulate a mapping from those conceptual elements to the concrete elements that represent them. Moreover, we found that for the definition of consistency rules, it can be challenging to reach an agreement between different stakeholders. We find that developing the consistency checking approach therefore already has benefits during its creation, as a vehicle for facilitating discussions among engineers on how their project should be structured. Hence, the experiences shared in this paper can be useful for both companies and academics working with companies in their efforts to initially introduce consistency checks. The developed framework gives a starting point for discussions around consistency and a template for implementing a first simple approach.

8.2. Future work

The work in this paper is the starting point for improving consistency management in our industrial setting. Now that the proof-of-concept is completed and the initial results were found helpful, the next step is to extend the checks further in three orthogonal directions. The first direction is to introduce more rules that check consistency between the currently studied artifacts, to more completely cover the architectural guidelines of the company. The second direction is to include more artifacts, of which engineers expressed two of particular interest. Hence, we aim to add checks between interface specifications in the system model and manually created XML documents specifying them (from these XML documents a part of the code is generated), as well as add checks between VHDL code and hardware blocks in the system model. The third direction is to improve the presentation of the results. Since the initial evaluation has shown the potential benefit of these consistency checks, we aim to return to our initial idea and study the possibilities of extending the consistency checks towards a blended modeling (Ciccozzi et al. 2019) setup, where the software engineers could view -and edit- parts of the system model from inside their IDE. Moreover, by further studying the adoption of consistency checks, we aim to identify which of the gathered statistics is a good indicator for model-and-code consistency.

Acknowledgments

This research was supported by Software Center — www.software-center.se. The authors also thank the system and software engineers of the industrial partner for their time and input in our discussions.

References

Ali, N., Baker, S., O'Crowley, R., Herold, S., & Buckley, J. (2018). Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, *23*(1), 224–258. doi: 10.1007/s10664-017-9515-3

Buckley, J., Mooney, S., Rosik, J., & Ali, N. (2013). Jittac: a just-in-time tool for architectural consistency. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 1291–1294). doi: 10.1109/ICSE.2013.6606700

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019). Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233. doi: 10.1007/s10270-018-00713-w

Ciccozzi, F., Tichy, M., Vangheluwe, H., & Weyns, D. (2019). Blended modelling-what, why and how. In *2019 acm/ieee 22nd international conference on model driven engineering languages and systems companion (models-c)* (pp. 425–430). doi: 10.1109/MODELS-C.2019.00068

Demuth, A., Kretschmer, R., Egyed, A., & Maes, D. (2016). Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: an experience report. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 529–538). doi: 10.1109/ICSME.2016.50

Egyed, A. (2001). Scalable consistency checking between diagrams-the viewintegra approach. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)* (pp. 387–390). doi: 10.1109/ASE.2001.989835

Egyed, A. (2010). Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, *37*(2), 188–204. doi: 10.1109/TSE.2010.38

Egyed, A., Zeman, K., Hehenberger, P., & Demuth, A. (2018). Maintaining consistency across engineering artifacts. *Computer*, *51*(2), 28–35. doi: 10.1109/MC.2018.1451666

Feldmann, S., Kernschmidt, K., Wimmer, M., & Vogel-Heuser, B. (2019). Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. *Journal of Systems and Software*, *153*, 105–134. doi: 10.1016/j.jss.2019.03.060

Feldt, R., & Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey. In *Proceedings of the software engineering and knowledge engineering conference* (pp. 374–379).

Friedenthal, S., Moore, A., & Steiner, R. (2014). *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann. doi: 10.1016/C2013-0-14457-1

INCOSE. (2015). *Systems engineering handbook: A guide for system life cycle processes and activities, 4th edition*.

Jongeling, R., Carlson, J., & Cicchetti, A. (2019). 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). doi: 10.1109/SEAA.2019.00071

Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., & Carlson, J. (2020). Towards consistency checking between a system model and its implementation. In *International conference on systems modelling and management* (pp. 30–39). doi: 10.1007/978-3-030-58167-1_3

Kanakis, G., Khelladi, D. E., Fischer, S., Tröls, M., & Egyed, A. (2019). An empirical study on the impact of inconsistency feedback during model and code co-changing. *The Journal of Object Technology*, *18*(2), 10–1. doi: 10.5381/jot.2019.18.2.a10

König, H., & Diskin, Z. (2017). Efficient consistency checking of interrelated models. In *European Conference on Modelling Foundations and Applications* (pp. 161–178). doi: 10.1007/978-3-319-61482-3_10

Leblebici, E., Anjorin, A., & Schürr, A. (2017). Inter-model consistency checking using triple graph grammars and linear optimization techniques. In *International conference on fundamental approaches to software engineering* (pp. 191–207). doi: 10.1007/978-3-662-54494-5_11

Lehman, M. M. (1996). Laws of software evolution revisited. In *European workshop on software process technology* (pp. 108–124). doi: 10.1007/BFb0017737

Lucas, F. J., Molina, F., & Toval, A. (2009). A systematic review of UML model consistency management. *Information and Software technology*, *51*(12), 1631–1645. doi: 10.1016/j.infsof.2009.04.009

Muram, F. u., Tran, H., & Zdun, U. (2017). Systematic review of software behavioral model consistency checking. *ACM Computing Surveys (CSUR)*, *50*(2), 1–39. doi: 10.1145/3037755

Murphy, G. C., Notkin, D., & Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (pp. 18–28). doi: 10.1145/222124.222136

Ralph, P. e. a. (2021). Empirical standards for software engineering research. *arXiv:2010.03525 [cs.SE]*. doi: 10.48550/arXiv.2010.03525

Riedl-Ehrenleitner, M., Demuth, A., & Egyed, A. (2014). Towards model-and-code consistency checking. In *2014 IEEE 38th Annual Computer Software and Applications Conference* (pp. 85–90). doi: 10.1109/COMPSAC.2014.91

Staron, M. (2020). *Action research in software engineering*. Springer. doi: 10.1007/978-3-030-32610-4

Stevens, P. (2014). Bidirectionally tolerating inconsistency: Partial transformations. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 32–46). doi: 10.1007/978-3-642-54804-8_3

Stevens, P. (2020). Maintaining consistency in networks of models: bidirectional transformations in the large. *Software and Systems Modeling*, *19*(1), 39–65. doi: 10.1007/s10270-019-00736-x

Torres, W., Van den Brand, M. G., & Serebrenik, A. (2020). A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, 1–20. doi: 10.1007/s10270-020-00834-1

Usman, M., Nadeem, A., Kim, T.-h., & Cho, E.-s. (2008). A survey of consistency checking techniques for UML models. In *2008 Advanced Software Engineering and Its Applications* (pp. 57–62). doi: 10.1109/ASEA.2008.40

Vierhauser, M., Grünbacher, P., Heider, W., Holl, G., & Lettner, D. (2012). Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 531–545). doi: 10.1007/978-3-642-33666-9_34

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer. doi: 10.1007/978-3-662-43839-8

About the authors

Robbert Jongeling is a PhD student at Mälardalen University, Department of Innovation, Design and Engineering in Västerås – Sweden. His current research is focused on the industrial adoption of continuous model-based development and the consequent challenges of consistency management across evolving models and other development artifacts. You can contact the author at robbert.jongeling@mdu.se or visit https://robbertjongeling.com.

Johan Fredriksson received his doctorate from Mälardalen University in 2008 and is currently a system engineer and software architecture specialist at Saab AB. You can contact the author at johan.fredriksson@saabgroup.com.

Jan Carlson is a professor in Computer Science, specializing in Software Engineering, at Mälardalen University. His current research focuses on model-based software and systems development, addressing areas such as artifact consistency, model-level timing analysis, and the combination of model-based development and continuous integration practices. You can contact the author at jan.carlson@mdu.se.

Federico Ciccozzi is an associate professor in Computer Science at Mälardalen University. His research interests cover many aspects of automated software and software language engineering, with a focus on model-driven and component-based software engineering for real-time embedded systems. You can contact the author at federico.ciccozzi@mdu.se.

Antonio Cicchetti is an associate professor at Mälardalen Univesity. His research interests target component-based and model-driven software engineering in industrial settings, including continuous modelling and related challenges, notably model versioning, languages and model transformations engineering and maintenance, and multiview/distributed development. You can contact the author at antonio.cicchetti@mdu.se.