Towards Consistency Checking Between a System Model and its Implementation*

Robbert Jongeling[1 [0000−0002−1863−3987]], Johan Fredriksson[2], Federico Ciccozzi[1 [0000−0002−0401−1036]], Antonio Cicchetti[1 [0000−0003−0416−1787]], and Jan Carlson[1 [0000−0002−8461−0230]]

[1] Mälardalen University, Västerås, Sweden
<firstname>.<lastname>@mdh.se
[2] Saab AB, Järfälla, Sweden
<firstname>.<lastname>@saabgroup.com

Abstract. In model-based systems engineering, a system model is the central development artifact containing architectural and design descriptions of core parts of the system. This abstract representation of the system is then partly realized in code. Throughout development, both system model and code evolve independently, incurring the risk of them drifting apart. Inconsistency between model and code can lead to errors in development, resulting in delayed or erroneous implementation. We present a work in progress towards automated mechanisms for checking consistency between a system model and code, within an industrial model-based systems engineering setting. In particular, we focus on automatically establishing traceability links between elements of the system model and parts of the code. The paper describes the challenges in achieving this in industrial practices and outlines our envisioned approach to overcome those challenges.

Keywords: Model-based systems engineering · Consistency checking · Agile model-based development

1 Introduction

The engineering of complex systems requires a team of engineers, each having specific expertise and working on different artifacts. A commonly investigated challenge is the management of consistency between the many different artifacts describing various aspects of the same system; for instance, consistency between a system model and code in model-based systems engineering (MBSE). There is empirical evidence indicating that inconsistency feedback improves the performance of software engineers in scenarios where the code needs to be updated after a change in the model [14]. Furthermore, the literature is rich in approaches that deal with consistency between artifacts, each considering a set of requirements identified as necessary for industrial adoption; they are discussed

* This research is supported by Software Center https://www.software-center.se.

in Section 4. However, few studies apply their approaches to industrial settings. In this paper, we report on a work in progress towards creating and evaluating an approach for checking the structural consistency between a system model and its corresponding implementation.

2 MBSE in the industrial setting under study

There are many different ways in which MBSE is adopted in industry. We do not discuss them all, but describe in this section how the MBSE paradigm is practiced in the industrial setting we study in this paper. Nevertheless, note that the proposed approach in Section 3 is not limited only to the industrial setting described in this paper. The outlined MBSE way-of-working within the studied setting is based on well-established standards such as the INCOSE systems engineering handbook [24], Friedenthal's "a practical guide to SysML "[8], and the ISO42010 standard [10]. Hence, our findings are expected to be generalizable beyond the scope of this specific scenario under study.

Several system engineers and around a dozen software engineers are involved in a collaborative effort to design and implement those systems. In this collaboration, the system engineers are mostly concerned with capturing the intended system design in a system model. This model is the core development artefact and contains the structural and functional design of the system. The functionality is decomposed into function blocks and then further implemented by smaller components. The model also captures the allocation of those components to either software or hardware. Note that the refinement of the system model into implementation is a manual effort, the system model does not contain enough detail to automatically generate code from it.

2.1 Overview of the system model

The system model is described using SysML diagrams. Each function block is represented by a block in a block definition diagram (BDD) containing a combination of system components, supplemented by an interface described by SysML Full ports. Each system component is described by a block in the internal block diagram (IBD) corresponding to the function block it is contained in. A system component contains attributes, operations, and signal receptions in the form of three proxy ports (for input, output, and test). Each operation defined in the system component is defined with an activity diagram. Furthermore, each system component contains a state machine defining its behavior. A schematic overview of these meta-elements is shown in Figure 1.

We are primarily interested in checking the consistency between the software realization of system components as defined in the system model, and the code implementing it. In the system model, the software realization consists of software blocks. They are defined in BDDs and then elaborated in IBDs, which define interfaces between different software blocks and the protocols by which messages are sent between them. Like function blocks, both software blocks and hardware blocks also define their own interfaces as SysML full ports.

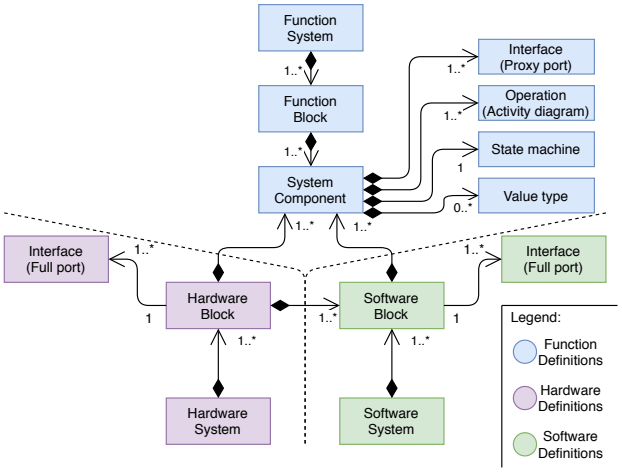

Fig. 1. Simplified overview of meta-elements composing a system model. Not depicted are the native, foreign, and test interfaces of function blocks.

2.2 Motivation for consistency checking

Development based on inconsistent artefacts can cause delays in development or worse, can cause the eventual implementation to be erroneous. For example, when the code violates interface definitions in the system model, some refactoring might be required to obtain the desired system. The longer such an inconsistency goes unnoticed, the more other code could be created that relies on it and therefore needs to be refactored upon eventual discovery of the inconsistency. Inconsistencies therefore need to be identified as early as possible and resolved before it causes harm. But complete consistency is also not possible, nor desirable, since the development should also not be inhibited. Awareness of inconsistencies soon after their introduction allows developers to decide the best course of action [12]. Therefore, we aim to detect, but not automatically repair, introduced inconsistencies between the system model and its corresponding implementation.

In the industrial setting under study, software engineers typically do not view the system model directly. Rather, changes to the system model are gathered by system engineers and presented to software engineers during handover meetings. These presentations, together with documents generated from the system model, serve as the primary input for software engineers to work on the implementation. At the same time, the system engineers have a limited view on what parts of the system model are implemented where in the code. The lack of traceability between system model and code complicates collaborative development, a lot.

Among other things, this lack of traceability inhibits development in shorter cycles [11]. The main concern, though, is the lack of impact analysis and thereby the potential late discovery of errors. Indeed, system engineers have, in this setting, limited support to assess the impact of potential changes in the system

model to related code, and late discovery of errors may induce large repair efforts. Hence, we aim to improve the communication between system engineers and software engineers, by providing inconsistency feedback to them both during their development activities.

2.3 Challenges

A model-code consistency checking approach requires two ingredients: i) traceability or navigation between model and code elements, determining which part of the model is realized by which part of the code, and ii) consistency rules that express what it means for elements to be consistent [20]. The first challenge to identifying inconsistencies is the scale of the system model under consideration, which implies that manually creating traceability links is not feasible. Automating this task yields numerous other challenges. We have summarized them in Table 1, and will detail them in the remainder of this section.

Model The SysML system model in this case study consists of BDDs, IBDs, state machines, and activity diagrams. In total, a model of a complex system can consist of hundreds of system components defined across dozens of diagrams. The corresponding code-base is similarly large, consisting of dozens of code files and make files across several repositories. To establish traceability between the model and the code, we must first determine which parts of the model are implemented in the code. The number of required traceability links scales with the number of model elements. Furthermore, navigating the model and code base to find the appropriate elements is a labor-intensive task.

As described, the model addresses several concerns and not all of them are reflected in the code, as for instance the allocation of system components to hardware. Hence, our approach should only attempt to identify code elements realizing model elements that are expected to be realized in the code.

When comparing model and code elements, some amount of inconsistency is to be expected, since the system model typically aims to describe the final product, whereas the code naturally always represents the current state of the

Table 1. Challenges to consistency checking in our industrial use-case.

Model	Size Distribution over many diagrams Addresses multiple concerns
Code	Spread out over different repositories Names *mostly* similar to model Variety in implementation of same concepts
Evolution	Existing model and code base Model and code evolve throughout project Way-of-working should not be hindered Model aims to capture end product, code captures current state

implementation. Our goal is not to ensure complete consistency all the time, but rather to indicate inconsistency as something that might signal problems in an early stage.

Code The implementation of the system model is spread out over a large number of code files. Among these are also additional artefacts such as make-files, required for building the system, that could be utilized to find dependencies that might not be described in the system model. A common software engineering practice is to organize such a project by creating several repositories to separate common, re-used functionality from components describing functionality for specific systems. This division is only shown at the software level and is not represented in the system model. Consequently, some additional effort is required to locate the implementation corresponding to given model elements, in order to create traceability links.

To trace elements between the artefacts, we cannot rely solely on the names of the elements. Partially, this is due to the complexity and separate evolution of system model and implementation code. The names are not purposefully obfuscated, but names of code elements diverge from names of model elements. This can happen, among other reasons, due to coding standards, typical use of abbreviations, or simply the impossibility of having spaces within names of code elements. In this way a model element "Hardware Monitor" can become "hw_mon" in the code.

Evolution It is also important to note that both a system model and a large code base already exist and are evolved separately throughout system development. We are not starting from scratch, but rather aim to introduce traceability links between the existing development artefacts. Beyond aforementioned scaling concerns, this also means that any proposed approach should not inhibit existing development processes.

3 Approach

In this section, we outline our envisioned approach to automatically establishing traceability links between model elements and code, within the described development scenario. We plan to extend this approach to report on the level of consistency between the linked artifacts, too.

3.1 Automatically discovering traceability links

In establishing traceability links, we can distinguish between two stages: initial creation and maintenance throughout development. Below, we describe our envisioned approach to these two stages while keeping in mind the challenges as outlined in Table 1.

Creating traceability links in an existing MBSE project As described in Section 2, we target traceability between those parts of the system model that have a corresponding representation in the code. In the described setting, there is a high naming consistency between for example C++ *namespaces* and SysML interfaces. As another example, state machines contained in system components are typically implemented in distinct classes in the code, where each state is expected to correspond to a method inside these classes. The patterns are typical, but not necessarily followed throughout the implementation. Hence, we cannot rely on them exclusively but require some additional input.

We outline here our plan to match model elements despite their differences, following the previously mentioned example of linking "Hardware Monitor" to "hw_mon". A dictionary is established for the translation of common abbreviations to their full form. In this way, "hw" and "mon" can be understood as "hardware" and "monitor" respectively. On top of that, some rewrite rules are created to transform naming from models and code into a common format. An example need for such rules is to remove spaces from all names of model elements. Using these rules, we can translate "hardware monitor" to "hw_mon". Finally, name comparisons should be case insensitive. Now, we can link "Hardware Monitor" to "hw_mon".

By applying these rules, some elements may be mapped directly, because after rule application, their name is identical. However, also collisions might occur where several names are all equal or very similar. To deal with this, we envision our approach as going through the model "top-down", i.e., starting from function blocks and following the model hierarchy down to the functionality defined in it. In this way, we aim to drastically narrow the search scope for elements to be matched. Since we have knowledge of which types of model elements are likely matched to which types of code elements, we can furthermore be more confident to suggest correct links. We follow the hierarchy as outlined in Figure 1. In the next step, we look at matching C++ *namespaces* to interfaces described as SysML full ports. Most model elements lower in the hierarchy are then expected to be found within the linked namespace. In cases where this does not yield a result, we plan to apply the name similarity approach to a wider set of artifacts.

Maintaining traceability links when artifacts evolve To be useful, a set of discovered traceability links needs to be updated throughout the evolution of the system. Newly implemented functionality should be reflected by additional traceability links. Removed or refactored functionality can lead to obsolete or outdated links that need to be discarded or updated. Hence, we envision an updating mechanism to be executed after a newly committed change to the model or code.

Changes can break existing links, for example due to the renaming of one of the linked elements. In this case, we cannot automatically assume that the traceability link should be kept, so instead we reapply the discovery stage for the affected function block. We believe that, in our setting, it is better than updating

the link automatically, because a rename might indicate also the following of a new system design. In other settings, the other alternative may be preferable.

4 Related Work

Despite the recognition of the importance of consistency [10] and the large body of work on consistency checking topics, few empirical evaluations and industrial applications have been published [4]. Indeed, model synchronization is still considered a challenge to industrial adoption of MDE. Selic identifies the scale of industrial applications as one of the main challenges to overcome for a model synchronization approach to be applicable. In particular, the number of consistency links can be huge and then the effort to maintain them will be very high and thus at constant risk of being neglected in favor of more pressing issues [22].

We focus in this work not on automatically synchronizing the model and code, but rather on identifying inconsistencies arising during development and providing modelers and developers with insights into them. Many formalisms have been proposed for capturing consistency rules. From languages like OVL [15] and EVL [16], to graph pattern matching approaches [9], to triple-graph grammars [21] and triple-graph patterns [7]. It should be mentioned that some of these approaches go further than merely identifying inconsistencies and additionally aim to repair. Devising another such approach is not the focus of this paper. Instead, we focus on the first required ingredient for consistency checking. Given an existing system model and corresponding implementation, we aim to automatically create traceability links between elements of the system model and parts of the code.

In further discussing the related work, it is important to note that we are not considering *requirements* traceability. Rather, we discuss here some approaches dealing with traceability between heterogeneous artifacts. For example, linking (parts of) artifacts using XML [23]. Or more formal approaches, such as "semantically rich" links [18], meaning that the links are formalized and can be automatically validated. Dependencies between different artifacts can also be explicitly modeled to enable inconsistency detection and a better overall management of the development process [19]. Building on top of that, another approach proposes modeling the development process to better identify and handle emerging inconsistencies [5]. This approach seems most appropriate when applied from the beginning of a new development project. Whereas in our case, we want to identify inconsistencies in already existing system model and corresponding code base.

Other types of approaches consider applying information retrieval methods to discover traceability links, although such methods are prone to reporting a large number of false positives [17]. In our work, we hope to limit these by combining the purely syntactic name information with the semantic information of model element types. For example, in the studied setting, a class name is not likely matched with the name of a state in a state machine; rather, that state would be matched to a function name. The revision history of development artifacts

can be utilized as well to detect traceability links by assessing what elements have been changed together in the past [13]. In our scenario, this approach is less suitable, since there is a clear separation between development of the system model and the code.

The concept of megamodeling is aimed at establishing links between models and model elements [6]. Megamodeling proposed a global modeling management framework, aimed at applying modeling techniques to numerous, large, complex artifacts distributed throughout a development setting and expressed in disparate modeling languages [3]. An example implementation is the tool AM3[3], which supports the automatic generation of traceability links based on existing model transformations. In the remainder of our work we will benefit from the concepts explored in megamodeling, particularly in the area of maintaining traceability links throughout the system's evolution.

It is clear that automated ways of obtaining traceability links are valuable to MBSE or any software development projects [1]. Furthermore, means to be notified of violations of architectural consistency are desired [2]. Nevertheless, few approaches of automating the discovery of traceability links have been studied in industrial settings.

5 Conclusion and future work

In this paper, we have discussed a work in progress towards checking consistency between a system model and code in a large scale industrial MBSE setting. We have outlined the state-of-practice and argued that consistency checking would be beneficial for both system engineers and software engineers. Although we have not yet implemented and rolled out consistency checks, we have identified challenges to its implementation and ideas to overcome them. These are summarized in Table 1 and are related to the system model, code, and the development process. In conclusion, establishing traceability links between system model and code in a real industrial setting is far from a trivial task.

The continuation of this work consists of three phases. In the first phase, we plan to implement automated support for discovering traceability links between elements of the system model and the implementation code. This will build further on the envisioned approach as briefly outlined in Section 3. We expect to add additional means of discovering links once we get to the implementation phase. The second phase consists of ensuring the semi-automated maintenance of traceability links when the model or code evolves. For some cases, this may be trivial once a link has been established and therefore there indeed is a code element corresponding to the model element. For more elaborate structural checks, like possible interface violations in the code, however, it remains necessary to evaluate the discovered links. In the final phase, we aim to calculate, given these traceability links, the structural consistency between the linked elements. And then, we aim to visualize the discovered consistency within the system model.

[3] `https://wiki.eclipse.org/AM3`

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Systems Journal **45**(3), 515–526 (2006). https://doi.org/10.1147/sj.453.0515
2. Ali, N., Baker, S., O'Crowley, R., Herold, S., Buckley, J.: Architecture consistency: State of the practice, challenges and requirements. Empirical Software Engineering **23**(1), 224–258 (2018). https://doi.org/10.1007/s10664-017-9515-3
3. Allilaire, F., Bézivin, J., Bruneliere, H., Jouault, F.: Global model management in eclipse gmt/am3 (2006)
4. Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Multi-view approaches for software and system modelling: a systematic literature review. Software & Systems Modeling pp. 1–27 (2019). https://doi.org/10.1007/s10270-018-00713-w
5. Dávid, I., Denil, J., Gadeyne, K., Vangheluwe, H.: Engineering process transformation to manage (in) consistency. In: Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016). pp. 7–16 (2016)
6. Favre, J.M.: Towards a basic theory to model model driven engineering. In: 3rd workshop in software model engineering, wisme. pp. 262–271. Citeseer (2004)
7. Feldmann, S., Kernschmidt, K., Wimmer, M., Vogel-Heuser, B.: Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. Journal of Systems and Software **153**, 105–134 (2019). https://doi.org/10.1016/j.jss.2019.03.060
8. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language. Morgan Kaufmann (2014)
9. Herzig, S., Qamar, A., Paredis, C.: An approach to Identifying Inconsistencies in Model-Based Systems Engineering. Procedia Computer Science **28**, 354–362 (2014). https://doi.org/10.1016/j.procs.2014.03.044
10. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011(E) Systems and software engineering – Architecture description. Tech. rep. (Dec 2011). https://doi.org/10.1109/IEEESTD.2011.6129467
11. 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
12. Jongeling, R., Ciccozzi, F., Cicchetti, A., Carlson, J.: Lightweight consistency checking for agile model-based development in practice. Journal of Object Technology **18**(2), 11:1–20 (July 2019). https://doi.org/10.5381/jot.2019.18.2.a11, the 15th European Conference on Modelling Foundations and Applications
13. Kagdi, H., Maletic, J.I., Sharif, B.: Mining software repositories for traceability links. In: 15th IEEE International Conference on Program Comprehension (ICPC'07). pp. 145–154. IEEE (2007). https://doi.org/10.1109/ICPC.2007.28
14. 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. Journal of Object Technology **18**(2), 10:1–21 (2019). https://doi.org/10.5381/jot.2019.18.2.a10
15. Kolovos, D., Paige, R., Polack, F.: The Epsilon Object Language (EOL). In: European Conference on Model Driven Architecture-Foundations and Applications. pp. 128–142. Springer (2006). https://doi.org/10.1007/11787044_11
16. Kolovos, D., Paige, R., Polack, F.: Detecting and Repairing Inconsistencies Across Heterogeneous Models. In: 2008 1st International Conference on

Software Testing, Verification, and Validation. pp. 356–364. IEEE (2008). https://doi.org/10.1109/icst.2008.23
17. Lucia, A.D., Fasano, F., Oliveto, R., Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. ACM Transactions on Software Engineering and Methodology (TOSEM) **16**(4), 13–es (2007). https://doi.org/10.1145/1276933.1276934
18. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. Software & Systems Modeling **10**(4), 469–487 (2011). https://doi.org/10.1007/s10270-010-0158-8
19. Qamar, A., Paredis, C.J., Wikander, J., During, C.: Dependency modeling and model management in mechatronic design. Journal of Computing and Information Science in Engineering **12**(4) (2012). https://doi.org/10.1115/1.4007986
20. Riedl-Ehrenleitner, M., Demuth, A., Egyed, A.: Towards model-and-code consistency checking. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 85–90. IEEE (2014). https://doi.org/10.1109/COMPSAC.2014.91
21. Schürr, A.: Specification of graph translators with triple graph grammars. In: International Workshop on Graph-Theoretic Concepts in Computer Science. pp. 151–163. Springer (1994). https://doi.org/10.1007/3-540-59071-4_45
22. Selic, B.: What will it take? a view on adoption of model-based methods in practice. Software & Systems Modeling **11**(4), 513–526 (2012). https://doi.org/10.1007/s10270-012-0261-0
23. Service, G., Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: Xlinkit: A consistency checking and smart link generation service. ACM Transactions on Internet Technology **2** (11 2001). https://doi.org/10.1145/514183.514186
24. Walden, D.D., Roedler, G.J., Forsberg, K., Hamelin, R.D., Shortell, T.M.: Systems engineering handbook: A guide for system life cycle processes and activities. John Wiley & Sons (2015)