Considerations About Consistency Management for Industrial Model-Based Development

Robbert Jongeling
robbert.jongeling@mdh.se

Mälardalen University
Västerås, Sweden

Abstract

Model-based development of complex embedded systems commonly leverages multiple models to describe systems at different levels of abstraction and from different viewpoints. Inconsistencies between these models may cause delays or late changes throughout system design, development and maintenance. Therefore, an efficient model-based development practice requires support for managing consistency between different models. This is complicated in practice because these models are possibly created in several different modelling languages and modelling tools. Furthermore, in order to be suitable for industrial adoption, consistency checking support must fit within current development processes and environments. This extended abstract describes a model-based development scenario from which a need for consistency checking emerges, followed by a discussion of requirements for consistency checking from a development process point of view. We also include a brief summary of early work on a prototype tool addressing these requirements.

1 Introduction

In model-based development (MBD), models are used as primary development artifacts in the specification, design, and implementation of systems [13]. For sound development, multiple models representing the same system should not contradict each other, i.e., they should be consistent. We refer to these models as *heterogeneous* since they may be expressed in different modelling languages and created in different tools. Keeping consistency between heterogeneous models is one of the challenges towards more industrial adoption of modelling practices, as commonly agreed on in the literature [10, 11, 12].

Complete consistency throughout development is not possible and not desired, since that would only hinder development [4]. Therefore, rather than ensuring consistency, the focus of a consistency checking approach should be to detect inconsistencies and notify developers of them. A plethora of consistency checking approaches have been proposed, mostly presenting technical frameworks that allow automatic detection of inconsistencies and sometimes also their automatic resolution.

Many of these approaches are demanding because the user needs to define and maintain many complex consistency definitions. Furthermore, they may require heavy changes to the development process and thereby discourage industrial adoption. To promote industry usage of a proposed consistency checking approach it is imperative that its design considers this process view [14]. In particular, usability of the approach for the target industry users. This extends to other factors such as the time and place of defining, maintaining, executing and resolving consistency checks in the development process.

In this extended abstract, we focus on what creating a consistency checking approach suitable for adoption in our target industrial contexts entails. We consider several aspects of the process view of consistency checking approaches and then summarize earlier results presenting a lightweight consistency checking approach in a specific MBD scenario. Creating such an approach requires us to find answers to the following

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.org/

research questions:

1. What are the types of inconsistencies users should be notified about?

2. What lightweight mechanism can detect the required inconsistencies?

3. How can this mechanism be made to fit in with existing development processes and environments?

2 Model inconsistencies in industry

In this section, we describe a model-based development scenario and the accompanying challenge of keeping heterogeneous models consistent. Using a running example, we then discuss the different aspects of a consistency checking approach designed to overcome this challenge.

2.1 Model-Based Development Scenario

In this scenario, we consider the model-based development of software for complex embedded systems. Models are used in the design and specification of the system and can be roughly divided into categories of high-level and low-level models. High-level models, e.g., SysML, AUTOSAR, or AADL models, describe the system design or architecture. Low-level models describe the software implementation, e.g., models in Simulink or executable UML. In some cases, development of the software implementation is done directly in code. Therefore, we also consider the scenario of the low-level models being code, e.g., C/C++. In general, we can describe the high-level and low-level models as pertaining to the system view and software view of the system respectively. In this scenario, the high-level model does not contain detailed enough information to enable automatic generation of code or low-level models. Models of both levels describe the same system but do so at different levels of abstraction. It is said that the low-level models are a *refinement* of the high-level models [12].

Adopting modern development practices, the models are developed in parallel and in short iterations. Work on high-level and low-level models thus happens in parallel and small increments. Typically, different engineers are responsible for the creation of the different models and they are possibly located in different teams and at different locations.

To illustrate the different aspects of the problem and to sketch the proposed approach towards a solution, we use a running example of a specific MBD scenario throughout the remainder of this paper. Consider the development of the braking system of a car. As part of the system model, there will be a SysML description of the anti-lock braking system (ABS), expressed in structure diagrams. This part of the system is then further refined and implemented in a Simulink model, which contains a detailed model for simulation and code generation. Now, consider the situation in which a design change to the ABS is made. In particular, we consider a change that makes the high-level system model and the low-level (Simulink) model inconsistent, for example, because of a difference in the assumed values for the car's total weight, or the amount of grip of the wheels.

2.2 Consistency Checking

We now describe the scope of the consistency checking challenge in the aforementioned MBD scenario. Multiple aspects of the challenge are considered. First, why inconsistency detection is needed and what inconsistencies an approach should be able to detect. Then, how inconsistencies can be detected. The method or mechanism by which inconsistencies are discovered is only a first part of a complete approach towards managing inter-model consistency. An often-overlooked aspect of proposed consistency checking approaches in the literature is its place in the engineering process. When, by whom and where the consistency checks should be created, maintained, executed, and resolved in the development process are also important aspects to consider for an approach to be suitable for industrial adoption. Any consistency checking approach should be designed with also these aspects in mind. They are also each discussed in this section.

2.2.1 Why do we need consistency checking?

Consider a single high-level model and several low-level models refining it. Let the models be consistent after iteration i_h of the high-level model and iteration i_l of the low-level models. When the high-level model undergoes iteration $i_h + 1$, it is likely that the low-level models, in i_l, are no longer consistent with it. For example, new functionality is added that is not yet implemented in low-level models, or existing function definitions are changed, requiring updates to the low-level models. In our running example, the high-level model is the system model expressed in SysML.

Each of these inconsistencies should to some extent be allowed, since disallowing inconsistency completely is detrimental to development [4]. On the other hand, if an inconsistency goes unnoticed for too long, it may cause severe problems. In general, in engineering, the earlier problems are detected, the cheaper they are to resolve. This is no less true for inter-model inconsistencies. When discovered late they can incur late changes and delays to development. When they are

not discovered at all, they can result in incorrect implementation.

For example, consider a change in the high-level model requiring a number of changes and additions in low-level models. In our running example, this indeed happens, the system model is changed, requiring some changes in the Simulink model. The inconsistency is introduced immediately after the changes to the high-level model and should be tolerated, at least temporarily. This is important since the system design can be created ahead of the software design. They do not need to be created simultaneously, so forbidding them would hamper development. But at some point the inconsistency should be resolved in order to prevent propagation of inconsistencies through multiple models used to describe the system. For example, when required changes to one of the low-level models are not yet made and other low-level models start relying on this, now inconsistent, low-level model. In the running example, not updating the Simulink model of the ABS might result in its incorrect implementation. A consistency check would have shown the inconsistency between the models after the change to the high-level model and in general, throughout development. It would also have given the developers a chance to resolve it before depending on the inconsistent low-level model and thus introducing more inconsistency.

2.2.2 What types of inconsistencies should be checked for?

In the described scenario, high-level models do not include low-level (software design) information such as algorithm descriptions. Rather, the similarities between the models at the different levels are structural. For example, part of the high-level model may define the structure of the required software, but with its most detailed elements being "black boxes", representing some particular functionality that is described in more detail in refining low-level models. Thus, the models are related, since they describe the same system but at different levels of abstraction.

In this scenario, a type of inconsistency helpful to know about is between model structures, i.e., the low-level models should indeed contain representations of what is described in the high-level models, even if that description is just a black-box. For example, consider a high-level model containing a definition for several software interfaces. Then, inconsistencies can exist when the low-level model implements more than, less than required, or different from the specified interface. Note that a difference can be seen as a combination of implementing less than required (the correct thing is missing) and more than required (the incorrect thing is done in place of the correct thing). The type of consistency checking in this setup is commonly referred to as vertical inter-model consistency checking [6].

Another type of inconsistency regards shared values between models. In the running example, if the system model and Simulink model assume a different total mass of the car, this inconsistency could result in an incorrect ABS. Several other types of inconsistencies exist and could be relevant to detect, we plan an industrial empirical study to derive a set of most relevant consistency types and scenarios.

2.2.3 How can inconsistencies be detected and resolved?

In the running example, a consistency check between the SysML model and the Simulink model could have detected the introduced inconsistency. Such a check should have been defined earlier in the development process and be automatically evaluated upon changes to either model.

There are numerous published approaches that allow for automatically detecting inconsistencies and sometimes also automatically resolving them. A considerable portion of work on consistency checking has focused specifically on UML models [9]. In our MBD scenario, we are not limited to UML but consider checking consistency between models in different modelling languages and tools.

Essentially, detecting inconsistencies between two different models requires two pieces of information. The first piece indicates the specific model elements that should be compared to each other and the second piece defines when those compared elements should be considered consistent. When defined in enough detail, for example, by using Triple Graph Grammars [2] or Link Models [3], the second part can be used to automatically resolve inconsistencies. This has the downside of needing a mapping between any two metamodel elements that need to be compared. To alleviate this, other approaches convert models to a common notation and detect inconsistencies in this common format, for example, through graph comparisons [5]. Another type of approach to detecting inconsistencies evaluates rules over multiple models. These rules can be expressed in, e.g., EVL [8], but possibly in any language that allows that can be automatically evaluated [1]. When considering applications in industrial contexts though, an important factor is the maintenance effort that consistency checks themselves introduce. If a consistency checking mechanism is heavy and labour-intensive to set-up and maintain, it is not likely to be used in industrial practice.

Several approaches have aimed at automatically resolving detected inconsistencies, e.g., in the context of inconsistencies between UML diagrams [15]. In our

development scenario, automatic resolution of inconsistencies is improbable. Firstly, because inconsistencies might indicate missing information which has to be newly created through manual implementation (a black-box can of course not be implemented automatically). Secondly, because it cannot always be determined which of the models should be changed if two are inconsistent. Sometimes functionality should be added in the one model and other times it should be removed from the other.

2.2.4 When to define and execute consistency checks?

Enabling the automatic detection of inconsistencies requires some initial definition of what elements to check and what to consider consistent. Depending on the approach parts of the initial definition could be made prior to development, or be implicit. For example, a logical rule could be defined stating that each class in a UML class diagram must be represented in at least one UML sequence diagram in the same model. This rule implicitly contains the specific mapping between model elements as each pair of class diagrams and sequence diagrams. When creating rules pertaining to specific model elements, they can typically not be created completely in advance. The definition of new consistency checks should therefore be possible throughout development.

During the system development, the design and implementation undergo numerous iterations, in each of which inconsistencies can be introduced and resolved. Furthermore, newly created model elements may incur the need for new or changed consistency checks. Therefore, the initial definitions should be able to be maintained, i.e., changed, added to, or removed from. The purpose of this maintenance is then to sharpen consistency check definitions and make their results more relevant.

Another aspect then concerns when in time to *execute* the consistency checks. Although several authors have argued for executing consistency checks continuously (e.g. [1]), this might not be suitable for a development scenario in which inconsistencies are unavoidable, since it would generate a large amount of trivial false positives, e.g., notifying the engineer of an inconsistent refinement immediately after creating the model element in the high-level model. The other extreme is naturally bad as well: when consistency checks are executed too infrequently, they cannot notify developers of inconsistencies early. An example middle ground is providing checks that are able to be executed on-demand by developers. Such on-demand checks might lead to missed inconsistencies, for instance, when they are not integrated well in the development process and therefore not executed frequently enough. A check at regular intervals in the process safeguards this problem while keeping the balance between being executed too frequently and not frequent enough. An example time to execute consistency checks is after each completed development iteration, in a continuous integration (CI) process. In our running example, this would detect the inconsistency between the models when the changes to the system models are integrated, avoiding many trivial notifications of inconsistency during the making of those changes.

Finally, we must consider when to *resolve* identified inconsistencies. This is very much related to the time of detecting the inconsistencies, and the person responsible for resolving them. Furthermore, acceptable resolving times depend on the specific project and the further development plans. In general, inconsistencies should be resolved as soon as possible, but depending on the exact situation, it may be acceptable to leave some inconsistencies exist for a longer time. In either case, this is up to the developers and an approach should appropriately support either choice.

In our running example, the definition of consistency checks should have been done prior to this occurrence of the inconsistency (or it would not be detected). The execution should be done some time after the introduced change to the model. As argued above, there is a fine balance between too frequent and too infrequent checks. The resolution of the inconsistency is then up to the developer, that can now decide a best time to resolve the problem before it spreads to other components.

2.2.5 Who defines and maintains consistency checks and who acts on detected inconsistencies?

Apart from the time of definition, maintenance and execution of consistency checks, of course an approach should consider which developers, in which roles, will create the checks. Defining sensible consistency checks requires knowledge of models at both abstraction levels. This implies the need for an approach providing these checks to be accessible to any involved developer in the project.

An important additional aspect concerns the responsibility of resolving detected inconsistencies. It is likely that an inconsistency caused by changes in the high-level should be resolved by changes in the low-level models. But this is not necessarily true. Changes in the high-level model might trail changes in the low-level models and therefore need updating. In the running example, consider instead that the Simulink model was updated first to reflect a redesign of the

system. Now it is inconsistent with the system model, but we would not want to revert the changes just made to the Simulink model in order for them to be again consistent. In either case, it is likely that the inconsistencies found by making changes at one level should be resolved at the other level and therefore by another engineer or group of engineers. In order not to raise more need for communication than it helps remove, a consistency checking approach should allow developers from both abstraction levels to view and understand results of consistency checks.

2.2.6 Where should consistency checks be defined and executed?

Similar to the when and who, an important aspect to consider is where in the development environment to define, maintain, and view results of executed consistency checks. To promote industrial adoption, a consistency checking approach should be accessible to all developers at all times throughout development. At the same time, it needs to fit in with existing development processes and tooling environments.

Indeed, including a consistency checking approach within an existing complex tooling environment is not straightforward. Integrating a new separate tool, apart from the technical challenges of interacting with the other existing tools, risks never being looked at since it is not part of any existing development workflow. Another placement of a consistency checking approach could be as a plug-in or extension to an existing modelling tool. While integrating an approach in one of the modelling tools has the advantage of developers using it without leaving their development environment, access to consistency checks is limited to users of that tool. This downside can be resolved by integrating consistency checks in a tool that is already part of the development process and accessible to all potential users, e.g., a version control system, an issue tracker, or a continuous integration (CI) server.

2.3 Requirements

We aim for a pragmatic approach to consistency checking that is applicable in industrial practice. The essence of any approach is to notify developers of detected inconsistencies. Furthermore, an approach suitable in practice requires considering the process aspects of MBD in industry. Given the described MBD scenario in Section 2.1, a consistency checking approach is required that is generic, i.e. applicable to models conforming to various different modelling languages. Additionally, the consistency aspects discussed in Section 2.2 show the need for an approach to consistency management to be *lightweight*, i.e., easy to use and minimally interfering with existing development processes and environments.

3 Proposed Approach

In this section we describe previously published early research results [7], as well as planned continuation and evaluation of that work. The early results present a lightweight consistency checking approach suitable for a scenario similar to what is described in Section 2, focusing on the structures of the high-level and low-level models.

3.1 Overview

Our early results focus on checking vertical inter-model consistency between heterogeneous models [7]. In particular, the approach allows checking for structural equivalence, i.e., checking that two model elements have the same structure, and structural refinement, i.e., one model element contains at least elements corresponding to similar ones as represented in the other model. It thus focuses only on the structural type of inconsistencies and not yet on other possible types. To this end, a common tree structure is created through model transformations. This tree represents the structures of the involved models. The approach focuses on being lightweight in use through a separation of concerns in the definition of consistency checks. Specifically, the two parts of our consistency checks are: 1) a global mapping between meta-model elements of different modelling languages, and 2) a user-defined mapping between model elements of different models. The "lightweight-ness" of the approach is ensured by having the user only being concerned with part 2: indicating which model elements across heterogeneous models should be consistent. Given this indication, checks are generated that detect inconsistencies throughout evolution of the involved models.

The two mappings are called a language consistency mapping (LC_{map}), and a model consistency mapping (MC_{map}). A LC_{map} describes how meta-model elements of different modelling languages are mapped to a tree structure and what elements are not mapped to it. For example, subsystems in Simulink models are mapped to nodes in the tree but more detailed blocks, for example describing mathematical operations, are not. In this way, the LC_{map} is global. It is defined once and reused for all consistency checks. A consistency check compares the trees created from models conforming to a particular language by invoking the corresponding LC_{map}. Note that the representation of models in the common format in comparison between common formats reduces the number of required mappings between metamodels. In a scenario where direct transformations exist between any two metamodels,

for a total of n metamodels, $\frac{n(n-1)}{2}$ (bi-directional) transformations are required, whereas in this case, n suffice.

A MC_{map} defines the specific model elements in specific models between which consistency should be checked. Consider a SysML model, and a set of Simulink models refining different parts of it. To create a consistency check for this case, the user selects a block in the SysML model and a Simulink model and indicates what type of consistency should be checked between them (refinement in this case). This constitutes the MC_{map}.

A complete run of a consistency check then consists of the following steps:

1. A MC_{map} between element e_1 of model M_1 in language L_1 and a model element e_2 of model M_2 in language L_2 is evaluated.
   (a) The LC_{map} between L_1 and L_2 consists of two defined transformations, building tree representations T_1 from M_1 and T_2 from M_2.
   (b) Now, T_1 and T_2 contain e_1 and e_2 respectively. Let T_{e1} and T_{e2} be the subtrees starting at e_1 and e_2 respectively.
2. Now, a comparison algorithm is executed (depending on the choice between refinement and equivalence in MC_{map})
   (a) The algorithm compares T_{e1} and T_{e2}.
   (b) The result is pass or fail. In case of fail a detailed description of the difference between T_{e1} and T_{e2} causing the comparison algorithm to report a failure.
3. The user sees the result of all evaluated MC_{maps} and can modify their definitions, including options to mute the check (do not show unless the result changes) and skip the check (do not show until re-enabled). Of course, MC_{maps} can also be deleted and new ones can be added.

The approach is implemented as a plug-in for Jenkins[1], an automation server often used for CI pipelines. This is a design choice partly motivated by the requirements as outlined in Section 2. The placement in the CI pipeline allows for frequently defining and maintaining of multiple MC_{maps}. Furthermore, consistency checks are automatically executed after each integration, when models are typically internally consistent, thus avoiding very temporary and other uninteresting inconsistencies from being reported. Running checks at each integration seems like a reasonable frequency, not too frequent and therefore tedious but frequent enough to detect meaningful inconsistencies early. When already in place in a project, adding the plug-in to the CI pipeline entails minimal overhead. It is also a place independent of modelling tools and centrally accessible for viewing results and maintaining checks. Since our approach does not aim to automatically resolve detected inconsistencies but rather at making developers aware of them, it is important that results of checks are easily visible.

[1] https://jenkins.io

3.2 Planned generalizations

The approach presented in [7] is focused on finding structural differences between high-level and low-level models. Its prototype implementation focuses in particular on SysML and Simulink models. Note that the approach does not depend on the chosen modelling languages. It also does not depend on the chosen intermediate tree structure as a representation for the models. To allow for a greater set of detectable inconsistencies, a more generic data structure can be chosen to represent the models. For example, models could be expressed as graphs, allowing checking of relationships between model elements other than hierarchical ones. In more general terms, we aim to design a metamodel allowing for expressing elements of many heterogeneous models, such that they can be compared and consistency can be checked between them. Thus, different formalisms can be envisioned as defining the LC_{maps}, the core idea of the approach is separating the consistency check definition in LC_{map} and MC_{map} and requiring the end-user only to define a very simple MC_{map}.

To make the approach even more lightweight, we envision using higher-order transformations (HoT) to derive the LC_{map} transformations from similar user actions as used to define the MC_{map}. In that case, the user-input to LC_{map} definitions would be limited to pointing to different modelling language concepts that can be compared to each other. The HoTs would then generate the model transformations required to represent the models in a tree, or other intermediate data structure.

3.3 Limitations

As said, the main limitation of the proposed approach is that it aims at comparing the structures of models. When we consider the generalized case, in which we have found a comparison metamodel, a remaining limitation is the local view on consistency checking. In essence, our approach is rule-based and comparing specific model elements.

Another fundamental limitation to our approach is that we limit ourselves to detecting inconsistencies, rather than attempting automatic resolution. We also

do not yet automatically propose ways to fix detected inconsistencies. Currently the limit of our approach is that it can provide a reason why the consistency check fails.

In terms of execution of the consistency checks, our approach assumes executing batches of checks at e.g. every integration. To limit the execution time, we aim to limit the amount of checks that are executed by running only those checks for which one of the involved models has changed since the last execution. Still, the approach is not checking for inconsistency in real-time, which is a design decision as explained in Section 2.

The strength of our approach with respect to existing approaches is mainly aimed to be that it is lightweight in usage. Where other approaches require complex rule definitions, for example in first-order logic or by means of bidirectional transformations, our approach requires of users only a model consistency mapping and a globally reusable language consistency mapping. Since the mainly considered scenario of applying this approach is in development and maintenance of complex systems, having a simple way of defining consistency checks is imperative.

3.4 Planned evaluation

In addition to the three research questions posed in Section 1, the required evaluation can be summarized in the following questions.

1. Can the proposed consistency checking approach detect the required inconsistencies?
2. Is the proposed consistency checking approach suitable for industrial practice?

The first question is commonly considered in related work. Almost all approaches, including our own as presented in [7], show an evaluation based on an example to showcase what inconsistencies can be detected. In their further evaluation, most publications focus on performance aspects such as computation time required and the scalability of their approach on larger models. These evaluations much less frequently include the usage of proposed consistency checking mechanism.

Some evaluations of approaches consider part of this process view. For example, Feldmann et al. [3] perform a small controlled user experiment to evaluate the ease of definition of consistency checks in their approach versus EVL. Egyed presents an approach that is user-friendly by being agnostic of the format of consistency rules, i.e., capable of handling consistency rules in any formalism [1]. In our proposed approach, the user does not define the semantics of consistency rules, but instead only points at model elements across heterogeneous models that should be checked for inconsistency.

An industrial evaluation of our approach requires a more mature tool and an implementation of the planned generalizations. The planned evaluation of our consistency checking approach then has two parts, corresponding to the two evaluation questions. First, we evaluate its suitability to detect the types of inconsistencies that should be detected. This evaluation is envisioned to be performed as a case study on a set of industrial models. The case study should show that the approach indeed detects the required inconsistencies and can do so in a reasonable amount of time, also when applied to models of scales commonly seen in industry. Second, an evaluation of the suitability of the approach in industrial practice is foreseen. Currently, we envision this evaluation as performing structured interviews with developers after they have been able to use our tool for some time.

As a first step towards performing this evaluation, we plan to work closely together with our industrial partners to get an answer to the question posed in Section 2: "What types of consistency should be checked?" So what types of inconsistencies occur in practice between what types of artifacts. The second step is then to investigate different ways of defining consistency checks and compare them with respect to the criteria we have specified here. An evaluation of the first question can then be done by using the tool on realistic industrial models. The evaluation of the second question can be done either by means of user experiments, or by evaluating the features of the approach with respect to requirements as defined by industrial practitioners, e.g. the requirement that the tool should be able to detect inconsistency between heterogeneous models.

4 Summary

Many published approaches consider the technical challenge of creating consistency checks with a focus on a specific set of requirements, but largely ignore the practical aspects of defining, maintaining, and executing them. In general, consistency checking work assumes the importance of checking consistency (why), explains the scope of the inconsistencies checked by their approach (what), and explains the mechanism devised to satisfy the approach (how). We have argued in this work the importance of additionally considering the engineers defining and maintaining the checks as well as responding to their results (who), the placement of definitions of the checks and execution in the development process (when), and the suitable place for tool support for defining and creating them (where).

In particular, we have considered three research questions towards a consistency checking approach suitable for industrial adoption. These questions have

been discussed in the context of a specific MBD scenario. Further, early work was presented towards lightweight consistency checking. The approach requires only the selection of model elements between which to check consistency and then uses a global consistency mapping between the modelling languages to generate and execute checks. For now, the approach is targeted to a limited scenario in which consistency between the structure of models is checked. In order for our consistency checking approach to be applicable in practice, we need a more definitive answer to the first two research questions, generalize our proposed approach to answer the third, and evaluate that approach to verify that it is indeed beneficial to the development process.

Acknowledgement

This work is supported by Software Center.[2]

References

[1] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.

[2] H. Ehrig, K. Ehrig, and F. Hermann. From model transformation to model integration based on the algebraic approach to triple graph grammars. *Electronic Communications of the EASST*, 10, 2008.

[3] S. Feldmann, K. Kernschmidt, M. Wimmer, and B. Vogel-Heuser. Managing Inter-Model Inconsistencies in Model-based Systems Engineering: Application in Automated Production Systems Engineering. *Journal of Systems and Software*, 2019.

[4] A. C. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[5] S. Herzig, A. Qamar, and C. Paredis. An approach to identifying inconsistencies in model-based systems engineering. *Procedia Computer Science*, 28:354–362, 2014.

[6] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille. Consistency problems in UML-based software development. In *International Conference on the Unified Modeling Language*, pages 1–12. Springer, 2004.

[7] R. Jongeling, F. Ciccozzi, A. Cicchetti, and J. Carlson. Lightweight Consistency Checking for Agile Model-Based Development in Practice. In *15th European Conference on Modelling Foundations and Applications (ECMFA)*, 2019.

[8] D. Kolovos, R. Paige, and F. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 356–364. IEEE, 2008.

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

[10] A. M. Madni and M. Sievers. Model-based systems engineering: motivation, current status, and needed advances. In *Disciplinary Convergence in Systems Engineering Research*, pages 311–325. Springer, 2018.

[11] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, et al. The relevance of model-driven engineering thirty years from now. In *International Conference on Model Driven Engineering Languages and Systems*, pages 183–200. Springer, 2014.

[12] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil. A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2013.

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

[14] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 329–380. World Scientific, 2001.

[15] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 315–324. ACM, 2009.

[2] `www.software-center.se`