Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Lightweight Consistency Checking for Agile Model-Based Development in Practice

Robbert Jongeling[a] Federico Ciccozzi[a] Antonio Cicchetti[a]
Jan Carlson[a]

a. Mälardalen University; School of Innovation, Design and Engineering; Västerås, Sweden

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

Keywords Consistency checking; Agile model-based development; Multi-view modelling.

1 Introduction

The Model-Based Development (MBD) paradigm holds the promise of improving productivity of the development process by promoting models as core artifacts, particularly in early development phases, i.e., specification and design [Sch06]. Further, models are also used for advanced development activities such as simulation and code generation. Besides, in industrial contexts, models as main project artifacts play an important role in documentation and communication between different development teams [ST18]. Models are becoming critical assets for development of industrial systems and software, not only within single projects but over several projects through model reuse. In modern industrial MBD practice, software systems are modelled through multiple views, using so-called *multi-view modelling* [CCP19].

Views are represented by *heterogeneous* models, i.e., models conforming to different modelling languages (often created with different tools, which complicates consistency checking). Usually, these views are exploited by different teams and for different aspects of development. Consider the context shown in Figure 1, where a system model, created by system designers to describe architectural matters, is refined into a set of software models by the software designers. In many cases, models across different views are closely related and they may partially overlap since they describe the same parts of a system. The use of multiple (often partially overlapping) views requires a careful checking and maintenance of consistency among them. Consistent models are in fact essential to ensure a coherent design as well as efficiency and correctness in the development process. While complete consistency (at any time in the development) may not be achievable or desirable, lingering inconsistencies can snowball into serious issues if not identified in early phases of development. A way to prevent this is to notify the developer about inconsistencies between models soon after their introduction, by means of consistency checking.

Consistency checking within a model (i.e. intra-model consistency), or between models conforming to the same modelling language, is often available in modelling tools. We focus on checking inter-model consistency between heterogeneous models, which is a more complex endeavour for several reasons. Firstly, inter-model consistency often

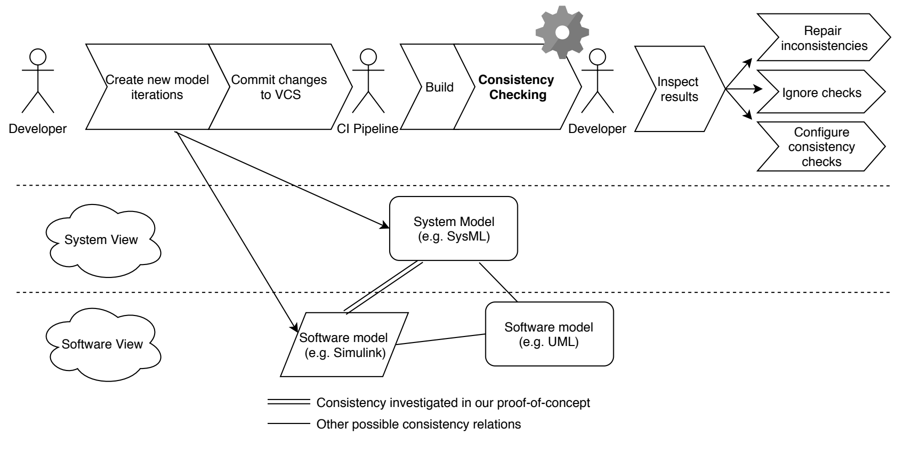

Developer
Create new model iterations
Commit changes to VCS
CI Pipeline
Build
Consistency Checking
Developer
Inspect results
Repair inconsistencies
Ignore checks
Configure consistency checks
System View
System Model (e.g. SysML)
Software View
Software model (e.g. Simulink)
Software model (e.g. UML)
Consistency investigated in our proof-of-concept
Other possible consistency relations

Figure 1 – Illustrating the scope of our approach, a tool performing inter-model consistency checking between heterogeneous models and notifying the developer about inconsistencies.

requires the ability to interact with a set of different modelling tools and processes in an industrial MBD context. Changes in this ecosystem are hard to make. Replacing a modelling tool to be able to perform inter-model consistency checks is often not feasible and any additional tool should not interfere with the existing ecosystem. Similarly, existing development processes are not easily changed, additional actions would be performed reluctantly in the best case, or skipped in the worst case, if they disrupt existing processes. Secondly, when consistency checks require a steep learning curve or excessive effort to create or maintain, the intended users may be discouraged from using them in the first place. Existing approaches, e.g. those based on Triple Graph Grammars [EEH08] or link-models [FWKVH16], are powerful but complex, hence requiring considerable effort to define and maintain consistency checks.

The context of this research is represented by an academia-industry collaboration called Software Center[1] and composed of 12 large companies and 5 universities. Among our industrial partners in Software Center, there is a clear trend of model-based development going agile. Very short cycles, typical of agile development, complicate consistency checking, especially if it requires a large effort in defining, maintaining and executing consistency checks. A consequence is that there is a need for a lightweight consistency checking approach. Lightweight means that it shall infringe minimally on existing development processes and tools, but aid developers in easily monitoring inter-model consistency. This kind of approach is currently lacking and much desired by practitioners.

In this paper, we show an application of consistency checking between heterogeneous models. We motivate requirements for a lightweight approach in Section 2, present a generic approach that satisfies these requirements in Section 3, and show an implementation of this approach in Section 4. Limitations and potential extensions to our approach are discussed in Section 5, a relevant portion of the extensive related work about consistency management is discussed in Section 6, while conclusions and some prospects of future work are included in Section 7.

2 Scope

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

2.1 Industrial context of consistency checking

Multi-view modelling refers to a practice in which a system is designed using multiple models (each of which representing a specific modelling view), potentially created in different tools and described by means of different languages [BBCW17]. Different models may describe the system under development, or just part of it, at different levels of abstraction and from different stakeholder perspectives, such as requirements engineer, system designer, or software developer. Yet, these models are commonly not disjoint, since they describe (parts of) the same system. There is often an explicit overlap, where multiple models describe, in the same or different levels of detail, the same parts of the system.

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

Kolovos et al. [KPP08] classify the relationships between models that induce this overlap, of these, the most relevant in industrial practice are "uses", "refines", "complements", "alternative for", and "aspect of". Due to the nature of these relationships, they are highly correlated to the structure of the models. Kolovos et al. [KPP08] go on to classify types of inconsistencies that can occur between overlapping models, the ones relevant to us are "incompleteness", "contradiction", "misuse", and "redundancy." Intuitively, a comparison of the structure of two overlapping heterogeneous models would show these types of inconsistency at a glance. While these relationships can occur between any pair of models, in our industrial context, we are primarily interested in consistency between models across different levels of abstraction, i.e., vertical inter-model consistency [HKRS05].

For example, let us consider a system model containing a SysML block B with two ports, P_1 and P_2. During system specification, the system designer might model parts of the system as a "black box", i.e., stop modelling at this level of abstraction and only care about the interfaces between blocks. Software designers on the other hand, as part of the system design, would model this as a "white-box", down to a more detailed level. They might, for instance, create a Simulink model S that describes B in more detail, with input and output ports corresponding to P_1 and P_2, and with additional details not included in B. This type of view relation between models S and B is commonly called *refinement* from S to B, or *abstraction* from B to S, respectively [PTQ$^+$13]. Other examples of these refinement relations include the one between a SysML model and an EPLAN[2] model to capture hydraulic schematics and between a SysML model and a Modelica model capturing the control system and dynamic behaviour, as exemplified in [SKSP10].

Figure 1 shows an overview of an industrial MBD context for which our proposed consistency checking is intended. Model inconsistencies across views, and thereby across e.g. specification, design, and implementation, complicate the development and evolution of systems. Inconsistencies shall never uncontrollably spread through the system design and one way to avoid this issue is by introducing consistency checks to support developers in identifying, at an early stage, possible inconsistencies in the system under development. Therefore, as shown in Figure 1, the development team is aided, during development and evolution of the architectural and software models, in keeping these models consistent through lightweight checks that indicate discrepancies in the structures of the created models. Note that in the different views, several heterogeneous models could exist, for example UML models in the software view (as shown in Figure 1). We highlight the generic applicability of our approach by choosing different languages in the example shown in Section 4.

To summarize, usable consistency checking, to ensure that models express overlapping concepts from different point of views without contradicting each other [PBO07], is pivotal for multi-view modelling approaches to be efficient. For industrial adoption, tool support is vital, too. Next, we elaborate on which requirements an industrial application of such a consistency checking mechanism entails.

2.2 Requirements

Since models conforming to different languages are typically designed using different tools and ensuring consistency is often a manual task, inconsistencies between them could remain unnoticed for considerable time during development. This is particularly

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

true when models are created in different views and for different aspects of the development. Let us exemplify in the context shown in Figure 1. During the specification and design of a car, a system model denotes the overall design of the car and more detailed models are designed to describe software, electronics, braking system, etc. A possible inconsistency could be introduced between the structural model, conforming to SysML, and the refining functional model, conforming to Simulink, that fails to refine a particular block of interest as defined in the structural model. We aim to support the checking of vertical consistency between heterogeneous models in cases where models are related by one of the aforementioned relations and a certain overlap in the structure of the models exists. As already mentioned, notifying developers of possible inconsistencies of this type is considered as very helpful in industrial practice, given the complexity of the systems and the distribution of the development efforts.

Overlaps causing possible inconsistencies are, in most cases, not one-to-one relations between entire models, nor between model elements at the same granularity level. Rather, since different models describe the same parts of the system at different levels of abstraction, the overlap is more likely to spread across the different levels of granularity, e.g. an entire model refining a subsystem, or a package of multiple blocks refining a model. For example, in the case of SysML and Simulink models describing the same system, a Simulink subsystem might not map one-to-one to a SysML block, but rather the SysML block might be refined via an entire Simulink model, containing several subsystems. Our approach allows the definition of consistency checks between related model elements across different languages and granularity levels. Since model elements may represent complex sub-models (a model element being the container root of a sub-tree of contained model elements), our approach should be able to recursively execute consistency checks too, to account for hierarchical compositions and containments across models.

The need for consistency checking becomes more pressing when companies adopt agile multi-view modelling, in particular, when the development includes continuous integration (CI). CI refers to the practice in which developers integrate their work frequently, multiple times per day, in a shared repository [FF06]. In this context, inconsistencies between heterogeneous models are easy to overlook but nevertheless important to identify as soon as possible, to prevent them from rapidly spreading to related artefacts. Agile development implies that models are developed in short iterations and in parallel with other models. Consequently, any of the overlapping models can be seen as anticipating changes in the others at any time during development, e.g., the system model may not yet contain concepts already described in software models and vice versa. In these settings, inconsistencies are inevitable and almost required, since forbidding them would hinder the concurrent and incremental nature of agile. Automatic resolution is undesirable too, since in most cases it can not be determined which of the involved models should be reconciled in a scenario where any can be anticipating the others. Moreover, temporary inconsistencies are sometimes required to allow for particular development activities [NEFE03]. Therefore, we want to allow developers to choose if and how to act on detected inconsistencies. For this reason, we propose a consistency checking approach that identifies and indicates inconsistencies to the developers, without enforcing their resolution.

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

Table 1 – Industrial practice (left) and the corresponding requirements (with ID) they entail (right).

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

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

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

Table 1 summarizes the requirements described in this section and their motivation. Our goal is to provide an approach, and tool support, for detection and notification of inter-model inconsistencies, across heterogeneous models and in a CI pipeline for agile MBD projects. We focus on structural equivalence between model elements or parts of models, as well as for structural refinement between model elements and parts of models.

3 Our consistency-checking approach

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

The types of consistency interesting for the developer depend on the involved modelling languages and the system under development. Hence, the meaning of consistency cannot be decided a priori, but should rather be specified by the person defining the consistency checks. In some existing consistency checking approaches, the meaning of consistency is captured in an intermediate translation, like a case by case dictionary, formally defining how to compare model elements between different models (and languages). An example of fixed medium to express these 'dictionary entries' is Triple Graph Grammars (TGGs) [EEH08]; this and other related mechanisms are discussed in more detail in Section 6.

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

Mappings between meta-model elements across different languages and the definition of the various kinds of consistency that can be checked (e.g., name equivalence) are described in *language*[3] *consistency mappings*. Mappings of model elements, across heterogeneous models, between which consistency should be checked and which specific kind of consistency to check are described in *model consistency mappings*. The user is only concerned with declaring and maintaining model consistency mappings, while the labor invested in creating language consistency mappings is limited to a one-time effort, unless the language undergoes changes. This makes the usage of our approach lightweight. Since we are dealing with heterogeneous models, in order to be able to compare them, and thereby check consistency, we need to represent them in a common notation.

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

3.1 Language consistency mapping

A language consistency mapping LC_{map} consists of:

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

(2) the definition of consistency types.

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

For example, in the case of Simulink models, blocks, subsystems and ports can be mapped to tree nodes, together with their hierarchical structure, whereas the operations inside blocks are not. Figure 2 shows an example of tree representation of a Simulink model, where “distiller” contains a subsystem “Distiller”, which in turn contains subsystems “Heat_Exchanger” and “Boiler”, which are in that hierarchy mapped to nodes in the tree.

Nodes inherit names from respective model elements and they are assigned an abstract type for comparison purposes (e.g., a Simulink inport and a SysML flowport become nodes of type ‘port’). Types can be leveraged to check consistency in cases

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

where name equivalence does not hold. For example, to check that two blocks, one in a SysML model and another in a Simulink model, contain the same number of 'ports', regardless of the names of these blocks and ports.

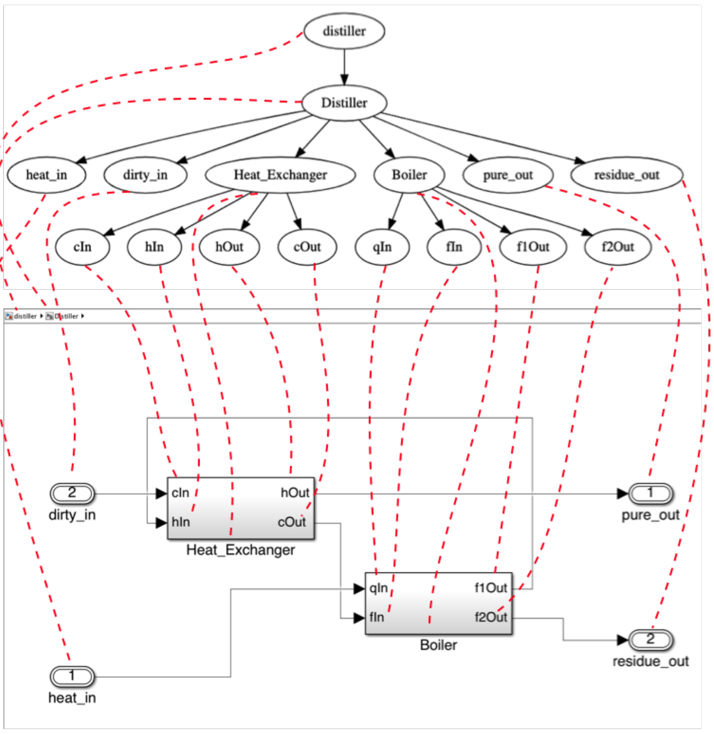


Figure 2 – Example of a transposition of a Simulink model to tree. Subsystems and ports are mapped to nodes in the tree, but not the simulation blocks inside the subsystems. The Simulink model is inspired by the well-known SysML Distiller example model [Hau06].

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

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

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

Figure 3 shows how a LC_{map} is used for comparing models. Technically, two models M_A conforming to L_A, and M_B conforming to L_B, are transposed into two corresponding trees T_A and T_B, conforming to TN, and comparisons are done between T_A and T_B.

Our proof-of-concept implementation provides two comparison rules, one for equivalence and one for refinement, exemplified in Figure 4. Since we can do comparison based on node names, types, and structure of their tree representations, we defined three levels of consistency *strictness*:

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

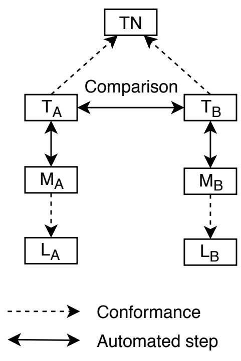


Figure 3 – LC_{map} consists of separate transpositions from both languages to a tree-based notation and a number of comparison rules. When executing a consistency check, automated model transformations transpose models into trees, between which automated comparison is run.

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

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

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

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

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

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

3.2 Model consistency mapping

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

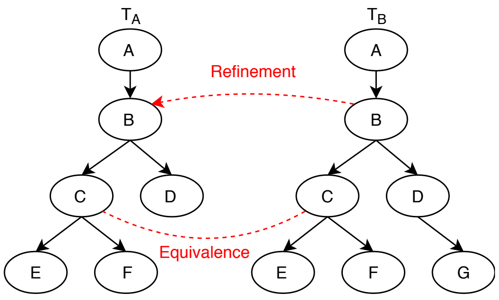


Figure 4 – Examples of an equivalence relation and a refinement relation between T_A and T_B. Node C in T_A is strictly equivalent to node C in T_B and node B in T_B strictly refines B in T_A.

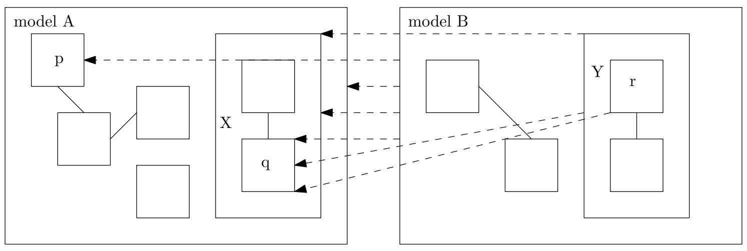


Figure 5 – Example of possible model consistency mappings between an abstract model A, refined by a model B. The dashed lines indicate possible refinement relations at different granularity levels.

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

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

As mentioned in the explanation of language consistency mappings, the tree-based notation allows to easily compare models and their elements, also when placed at different hierarchical levels. Figure 5 illustrates examples of model consistency mappings. For instance, model B could be a refinement of model A, or parts of it, such as sub-model X or elements p or q. Similarly, parts of model B, for example sub-model Y, could refine sub-model X or element q. Lower level mappings are possible too: for example element r in model B refining element q in model A.

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

3.3 Continuous integration pipeline

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

1. MC_{map} is evaluated. Consider a mapping between model element e_A of model M_A in language L_A and a model element e_B of model M_B in language L_B
   (a) LC_{map} between L_A and L_B is used to create trees T_A and T_B from models M_A and M_B, respectively.
   (b) In T_A and T_B, nodes corresponding to e_A and e_B are compared using a comparison rule, corresponding to a combination of the type of check (equivalence or refinement) and the strictness level (strict, intermediate, or loose). Since comparison rules define a comparison between nodes by including, recursively, their children, technically the subtrees with root nodes represented by e_A and e_B are compared.
   (c) The result of executing the CC is summarized as a binary outcome: pass or fail. In case of a failed check, a summary of the reasons behind the failure is shown to the user.
2. Configuration of existing model consistency mappings can be modified, including options to mute or skip checks in future runs.
3. The user can also add or delete model consistency mappings.

4 Proof of concept

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

In the remainder of this section, we show the process of defining and executing consistency checks on the Distiller example [Hau06] and applying it to one model consistency need that we identified exists in our industrial partners: a functional Simulink model refining a structural SysML model. Two models are created, one SysML model, shown in Figure 6, and one Simulink model which refines selected subsystems of the SysML model, and which was shown earlier in Figure 2.

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

[5]`https://jenkins.io/`

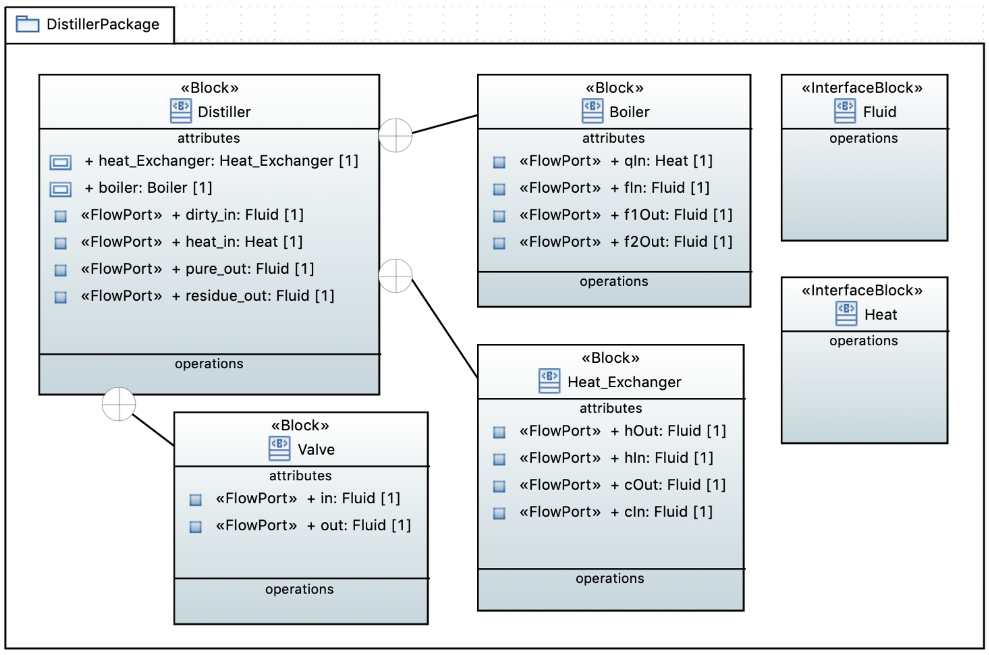

DistillerPackage
«Block»
Distiller
attributes
+ heat_Exchanger: Heat_Exchanger [1]
+ boiler: Boiler [1]
«FlowPort» + dirty_in: Fluid [1]
«FlowPort» + heat_in: Heat [1]
«FlowPort» + pure_out: Fluid [1]
«FlowPort» + residue_out: Fluid [1]
operations
«Block»
Boiler
attributes
«FlowPort» + qIn: Heat [1]
«FlowPort» + fIn: Fluid [1]
«FlowPort» + f1Out: Fluid [1]
«FlowPort» + f2Out: Fluid [1]
operations
«InterfaceBlock»
Fluid
operations
«InterfaceBlock»
Heat
operations
«Block»
Heat_Exchanger
attributes
«FlowPort» + hOut: Fluid [1]
«FlowPort» + hIn: Fluid [1]
«FlowPort» + cOut: Fluid [1]
«FlowPort» + cIn: Fluid [1]
operations
«Block»
Valve
attributes
«FlowPort» + in: Fluid [1]
«FlowPort» + out: Fluid [1]
operations

Figure 6 – Simplified block definition diagram of the SysML distiller example [Hau06].

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

4.1 Language consistency mapping

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

SysML. A subset of SysML diagrams is represented by structural diagrams, i.e., block definition diagrams and internal block diagrams. In this work, we focus on SysML models described in terms of these diagrams. In our tree-based notation, the root node represents the entire SysML model and the tree hierarchy reflects the structural hierarchy of the model. The root's children are packages or blocks. Packages can contain other packages and blocks, while blocks can contain other blocks and

[6]https://www.eclipse.org/xtend/

ports. The SysML model in the running example was created using Eclipse Papyrus[7]. The translation of the model to a tree is performed taking in input the `.uml` file, which contains the model definition (without diagrammatic information). In our transformations, we leverage the `EMF Ecore Resource` facilities to programmatically access the contents of this type of file.

Simulink. To parse Simulink models, from binary `.slx` or serialized `.mdl` format, we rely on CQSE's Simulink Library for Java[8]. As for the SysML model, the root node of the tree represents the Simulink model. The children nodes are then the SubSystems, Inports, and Outports contained in the models. SubSystems can contain other SubSystem, Inports and Outports. Note that we choose to omit certain types of blocks used to specifically implement Simulink simulations, such as logic operations and data conversions, since they do not affect the model structure.

4.2 A consistency checking tool

In this section we detail the approach steps enumerated in Section 3.

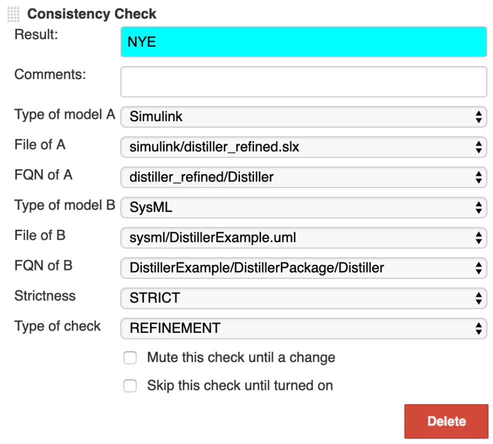
Defining model consistency mappings. Model consistency mappings are defined inside the Jenkins plug-in, by selecting the model elements between which consistency should be checked as well as the type and strictness of those checks. Figure 7 shows an example of consistency check definition. In our example, the type of model can be Simulink or SysML, but this can be extended to any language for which a transformation to the tree-based notation is implemented. When a modelling language is selected, the next drop-down box is populated with all model files of that language in the Jenkins workspace. When a file is selected, the next drop-down box is populated with all fully qualified names (FQNs) of model elements in the model, as represented in the related tree. Eventually, the strictness and type of check are selected. Note that, after checks are executed, the user can select to mute or skip them in future runs. Before executing the check, its result is set to *NYE* (Not Yet Executed), and no further comments are available.

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

Comparing trees. The first step in executing a consistency check is to transform the models to trees. Resulting trees for our running example are shown in Figure 8, where the black nodes represent the model elements to be compared (their selection in the MC_{map} can be seen in Figure 7. The refinement relation is now checked, not between the complete trees, but between the subtrees starting at the black nodes.

View results and manage configuration. In this case, the consistency check fails, since the model element in model A is not a refinement of the model element in model B. The `Valve` is in fact missing in the Simulink model as compared to the SysML model. This short explanation is shown in the result field of the MC_{map} definition, as shown in Figure 9. More detailed logs are available in the console output in Jenkins. A whole cycle of definition and execution of consistency checks is now

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



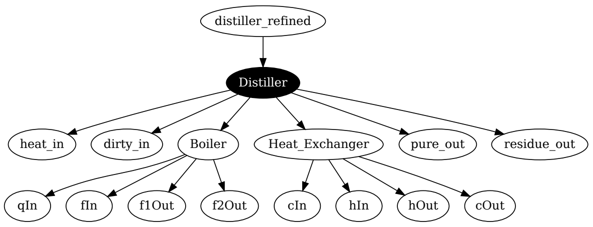
Consistency Check

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

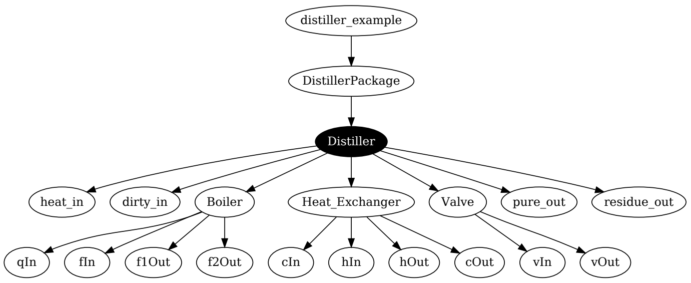
☐ Mute this check until a change

☐ Skip this check until turned on

Figure 7 – Example definition of a model consistency mapping in the Jenkins plug-in. Here, the element Distiller of the Simulink model distiller_refined is said to strictly refine the element Distiller in package DistillerPackage in the SysML DistillerExample model.



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



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

Figure 8 – Tree representations of the Simulink and SysML models; the subtrees with root nodes indicated in black are compared.

completed. New consistency checks can be defined and existing ones edited or deleted. Model consistency mappings can also be left unaltered to be run again in future builds, or set to be skipped or muted. When a check is skipped, it is not executed in future builds, until the user enables it again. When muted, a check is executed but its results are hidden, unless they are different to previous results in its previous execution. This allows the user to mute reports on inconsistencies that are relevant but temporarily tolerated, for example when modifying a model and before propagating the changes to other related models.

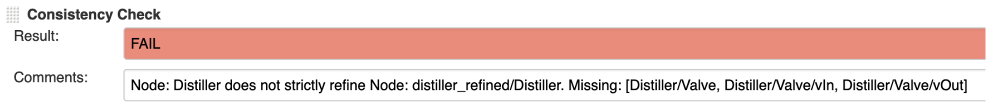

Consistency Check
Result: FAIL
Comments: Node: Distiller does not strictly refine Node: distiller_refined/Distiller. Missing: [Distiller/Valve, Distiller/Valve/vIn, Distiller/Valve/vOut]

Figure 9 – Result message of the failed strict refinement check.

5 Discussion

In this work, we have focused on lightweight consistency checking to help developers discover structural inconsistencies between heterogeneous models. In particular, we have considered the requirements (Rx) summarized in Table 1. R1-R2-R3 are satisfied by choosing to construct an abstract tree representation from models. Indeed, this allows checking between models in different languages, since we compare their representations in a common format, but more importantly, this format represents the structural characteristics of the models, enabling their comparison. Comparison rules are defined between tree nodes, regardless of their position in the tree, so they enable consistency checking between model elements at different levels of granularity, for example an entire Simulink model can be compared to a single SysML block. R4 is fulfilled by providing detailed feedback on detected inconsistencies to the user, but not automatically resolving inconsistencies. R5-R6, regarding frequent execution and minimal impact on the existing ecosystem of consistency checks, are satisfied by the implementation of our approach in a CI pipeline. This provides a natural environment for executing the defined checks frequently, while not requiring a particular modelling tool nor notable changes to the development process. R7 states that our approach should consist of consistency checks that are easy to create, use, and maintain. This is satisfied by separating language consistency mappings from model consistency mappings, requiring the user to only input a small amount of information to generate and execute consistency checks. These checks are defined once and executed at each integration, unless they are skipped, muted, or deleted by the user.

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

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

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

In the current implementation, we have focused on a specific example relevant to industrial practice. To perform a full-scale industrial evaluation however, requires the implementation to be enhanced with additional language consistency mappings and capabilities to check other types of inconsistencies.

6 Related work

Consistency among and within views is pivotal to ensure efficiency and correctness in the development process [ISO11]. This work provides an approach to lightweight consistency checking between heterogeneous models in a multi-view modelling context. In particular, we study an industrial multi-view modelling environment [ST18] in combination with agile development practices.

Dajsuren et al. [DGS$^+$14], also consider consistency between different views. Similarly to our approach, the authors prototype a tool for SysML structural diagrams aimed at the automotive industry, but the underlying approach is applicable to other languages as well. To enable comparison between models, both are first expressed at the same level of abstraction. The resulting models are compared as graphs to detect inconsistencies based on missing model elements or relations in one model that are declared in the other model. In their approach, model elements are annotated directly in the modelling tool to denote consistency between model elements at the same granularity level. Similarly, our approach aims to compare consistency between two different views with some structural overlap, but in addition it allows for checks across heterogeneous models and model elements at different granularity levels.

In this work, we create an abstract tree representation of models to enable comparisons between them. Other works employ other formalisms to achieve the same goal of being able to compare models in different languages. An often used mechanism is Triple Graph Grammars (TGGs), which allow a formal definition of the mapping of model concepts across different languages [EEH08], for instance between SysML and Modelica as done by Johnson et al. [JKPB12]. As opposed to our approach, these approaches require a high effort in declaring and maintaining the consistency checks.

The consistency checking approach proposed by Egyed allows for the creation of consistency rules in any formalism [Egy10]. Notably, in this approach, consistency checks are only executed when model elements they cover are changed, thus improving over approaches in which batches of checks are executed periodically. It can be a valuable future enhancement of our approach to similarly only execute those checks that relate model elements that have changed since the last execution.

Another means capturing the specific way of comparing particular model elements are link-models [FWKVH16]. These link-models declare the relation between parts of models, and constraints on that relation, by relating model elements through particular types of links, equivalence, refinement or satisfies. The link-models are then used to derive validation rules that can be automatically executed. The applicability of this approach is limited to MOF-based models, whereas our approach is meta-metamodel independent.

Similar to our approach, also graph structures have been proposed as an intermediate representation of models as well as the starting point for detecting inconsistencies [HQP14]. There, the graphs represent logical facts contained in the model, such that inconsistencies between graphs mean inconsistencies in the models. In our approach, the tree denotes not such logical facts, but rather focuses on the model structure.

In addition to approaches based on intermediate representations of models, others have proposed different means of comparison between models. For example, by declaring statements based on first-order logic to express facts that should be true about models [GFN02]. Later, these ideas were more matured and generalized, for example in the Epsilon Object Language [KPP06]. The advantage of our approach relative to these approaches is that the developer is not tasked with declaring such statements, since the meaning of consistency is captured in the language consistency mapping and the developer just specifies which model elements should be consistent.

The existing literature on consistency management in general is extensive [CCP19], so necessarily, the included works cover only a small portion of it. Notably, Feldmann et al. categorize existing approaches as proof theory-based, rule-based, or synchronization-based [FHK$^+$15]. Our approach can be categorized as synchronization-based, where the language consistency mappings define how model elements should be compared between languages, albeit not by a direct comparison but through an intermediary tree structure. Moreover, a plethora of approaches exists for consistency checking between UML models [LMT09]. Even though there are numerous approaches presented, we are not aware of any approach satisfying the requirements with respect to lightweightness as listed in Section 2.

7 Conclusions and future work

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

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

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

References

[BBCW17] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, Sep 2017. doi:10.1007/s10270-017-0622-9.

[CCP19] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software & Systems Modeling*, pages 1–27, 2019. doi:10.1007/s10270-018-00713-w.

[DGS$^+$14] Yanja Dajsuren, Christine Gerpheide, Alexander Serebrenik, Anton Wijs, Bogdan Vasilescu, and Mark van den Brand. Formalizing Correspondence Rules for Automotive Architecture Views. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, pages 129–138. ACM, 2014. doi:10.1145/2602576.2602588.

[EEH08] Hartmut Ehrig, Karsten Ehrig, and Frank Hermann. From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. *Electronic Communications of the EASST*, 10, 2008.

[Egy10] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2010. doi:10.1109/tse.2010.38.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[FHK$^+$15] Stefan Feldmann, Sebastian Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan Paredis, and Birgit Vogel-Heuser. A Comparison of Inconsistency Management Approaches Using a Mechatronic Manufacturing System Design Case Study. In *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 158–165. IEEE, 2015. doi:10.1109/coase.2015.7294055.

[FWKVH16] Stefan Feldmann, Manuel Wimmer, Konstantin Kernschmidt, and Birgit Vogel-Heuser. A Comprehensive Approach for Managing Inter-Model Inconsistencies in Automated Production Systems Engineering. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1120–1127. IEEE, 2016. doi:10.1109/coase.2016.7743530.

[GFN02] Clare Gryce, Anthony Finkelstein, and Christian Nentwich. Lightweight Checking for UML Based Software Development. In *Workshop on Consistency Problems in UML-based Software Development., Dresden, Germany*, 2002.

[Hau06] Matthew Hause. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12. Citeseer, 2006.

[HKRS05] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. Consistency Problems in UML-Based Software Development. In *UML Modeling Languages and Applications*, pages 1–12. Springer, 2005. `doi:10.1007/978-3-540-31797-5_1`.

[HQP14] Sebastian Herzig, Ahsan Qamar, and Christiaan Paredis. An approach to Identifying Inconsistencies in Model-Based Systems Engineering. *Procedia Computer Science*, 28:354–362, 2014. `doi:10.1016/j.procs.2014.03.044`.

[ISO11] ISO/IEC/IEEE Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46, Dec 2011. `doi:10.1109/IEEESTD.2011.6129467`.

[JKPB12] Thomas Johnson, Aleksandr Kerzhner, Christiaan Paredis, and Roger Burkhart. Integrating Models and Simulations of Continuous Dynamics into SysML. *Journal of Computing and Information Science in Engineering*, 12(1):011002, 2012. `doi:10.1115/1.4005452`.

[KPP06] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006. `doi:10.1007/11787044_11`.

[KPP08] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 356–364. IEEE, 2008. `doi:10.1109/icst.2008.23`.

[LMT09] Francisco Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631–1645, 2009. `doi:10.1016/j.infsof.2009.04.009`.

[NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):28–63, 2003. `doi:10.1145/839268.839271`.

[PBO07] Richard Paige, Phillip Brooke, and Jonathan Ostroff. Metamodel-Based Model Conformance and Multi-view Consistency Checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):11, 2007. `doi:10.1145/1243987.1243989`.

[PTQ$^+$13] Magnus Persson, Martin Torngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe, and Joachim Denil. A Characterization of Integrated Multi-View Modeling in the

Context of Embedded and Cyber-Physical Systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10. IEEE, 2013. doi:10.1109/emsoft.2013.6658588.

[Sch06] Douglas C Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[SKSP10] Aditya Shah, Aleksandr Kerzhner, Dirk Schaefer, and Christiaan Paredis. Multi-view Modeling to Support Embedded Systems Engineering in SysML. In *Graph transformations and model-driven engineering*, pages 580–601. Springer, 2010. doi:10.1007/978-3-642-17322-6_25.

[ST18] Jagadish Suryadevara and Saurabh Tiwari. Adopting MBSE in Construction Equipment Industry: An Experience Report. In *25th Asia-Pacific Software Engineering Conference APSEC*, 2018. doi:10.1109/apsec.2018.00066.

About the authors

Robbert Jongeling is a doctoral student focusing on industrial application of model-based development. His research interests include industrial adoption of agile MBD, consistency checking, and model evolution. Contact him at robbert.jongeling@mdh.se, or visit https://www.es.mdh.se/staff/3731-Robbert_Jongeling.

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

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

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

Acknowledgments This work is partially supported by Software Center[9] and by the Knowledge Foundation in Sweden through the MINEStrA project. The authors would like to thank Jagadish Suryadevara from Volvo Construction Equipment for his input in discussions about the work and his comments on early versions of the manuscript.

[9] www.software-center.se