The Journal of Systems and Software 153 (2019) 105–134

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering

S. Feldmann[a,*], K. Kernschmidt[a], M. Wimmer[b], B. Vogel-Heuser[a]

[a] Chair of Automation and Information Systems, Technical University of Munich, Boltzmannstr. 15, Garching near Munich 85748, Germany
[b] CDL-MINT, Institute of Business Informatics - Software Engineering, Johannes Kepler University, Altenbergstr. 69, 4040 Linz, Austria

ARTICLE INFO

Article history:
Received 21 January 2018
Revised 25 January 2019
Accepted 24 March 2019
Available online 25 March 2019

Keywords:
Automated production systems
Model-based systems engineering
Inconsistency management

ABSTRACT

To cope with the challenge of managing the complexity of automated production systems, model-based approaches are applied increasingly. However, due to the multitude of different disciplines involved in automated production systems engineering, e.g., mechanical, electrical, and software engineering, several modeling languages are used within a project to describe the system from different perspectives. To ensure that the resulting system models are not contradictory, the necessity to continuously diagnose and handle inconsistencies within and in between models arises. This article proposes a comprehensive approach that allows stakeholders to specify, diagnose, and handle inconsistencies in model-based systems engineering. In particular, to explicitly capture the dependencies and consistency rules that must hold between the disparate engineering models, a dedicated graphical modeling language is proposed. By means of this language, stakeholders can specify, diagnose, and handle inconsistencies in the accompanying inconsistency management framework. The approach is implemented based on the Eclipse Modeling Framework (EMF) and evaluated based on a demonstrator project as well as a small user experiment. First findings indicate that the approach is expressive enough to capture typical dependencies and consistency rules in the automated production system domain and that it requires less effort compared to manually developing inter-model inconsistency management solutions.

1. Introduction

The use of model-based approaches is a crucial and competitive factor for successful engineering processes in the automated production systems domain, and hence an emerging practice in industry (Schmidt, 2006). To represent the specific views of the system under design, e.g., requirements engineering, system specifications, software design or system analyses, a variety of different models is created within a project. To address their specific viewpoints stakeholders make use of various, heterogeneous modeling languages and tools (Broy et al., 2010). Although all models represent different aspects of the same system under investigation, dependencies between these models are inevitable (Vogel-Heuser et al., 2015; Spanoudakis and Zisman, 2001; Gausemeier et al., 2009). As a consequence, inconsistencies are likely to occur and have to be carefully considered to guarantee a high quality of the final system design.

To tackle this challenge we present a framework to enable stakeholders to

- define and elaborate links between the disparate, but overlapping engineering models,
- specify consistency rules – both within the engineering models and in between them,
- continuously diagnose potentially occurring inconsistencies, and
- handle the life-cycle of these inconsistencies through either ignoring, tolerating or resolving them.

We build on the basic approach presented in our previous work (Feldmann et al., 2016b), in which we discussed the usage of model patterns as the basis for relating models and inter-model inconsistency management. Based on our previous findings, we extend our approach with:

- an additional link type modeling language to define and elaborate the potential overlaps between engineering models in a concise manner, now also allowing for n-to-m links as well as to specify uniqueness constraints for the link ends,

* Corresponding author.
E-mail addresses: feldmann@ais.mw.tum.de, stefan.feldmann@tum.de (S. Feldmann), kernschmidt@ais.mw.tum.de (K. Kernschmidt), wimmer@big.tuwien.ac.at, manuel.wimmer@jku.at (M. Wimmer), vogel-heuser@ais.mw.tum.de (B. Vogel-Heuser).

https://doi.org/10.1016/j.jss.2019.03.060
0164-1212/

- an extended graphical, pattern-based modeling language to define the consistency rules, now also allowing for specifying different mathematical mapping properties such as distinguishing between relations and different function types,
- a semi-automatic inconsistency management framework, now coming with according prototypical tool support that allows for automatically generating executable consistency rules based on the graphically expressed model patterns,
- an evaluation that indicates first findings regarding the specification capabilities and efforts of our pattern-based modeling language as well as scalability of our framework for model-based engineering in the automated production systems domain, and
- first insights into the usability of our approach through a small user experiment.

The remainder of this article is structured as follows: Section 2 introduces the motivation behind our research, especially the challenges to be tackled by our inconsistency management framework. Subsequently, we introduce an application example in Section 3 with the according preliminaries. The inconsistency management framework is introduced in Section 4. By means of a prototypical implementation presented in Section 5 we provide first findings of evaluating our framework in Section 6 for an automated production system demonstrator project. The findings gained from this evaluation are underpinned through a minor user experiment; the procedure and results of this experiment are discussed in Section 7. Related research work is discussed in Section 8, before we conclude with an outlook on future work in Section 9.

2. Motivation and challenges

This section is devoted to the motivation and challenges of our research work. First, we introduce the motivation and objectives that arise from the engineering of automated production systems (Section 2.1). Second, we derive the challenges that must be tackled in order to address these objectives (Section 2.2).

2.1. Motivation

Due to the rising complexity of automated production systems, appropriate engineering processes are necessary that support engineers in developing their systems more rapidly and efficiently. Consequently, the use of models is a key methodology to develop and handle complex systems (Schmidt, 2006).

Motivated by the example of classical computer science, Model-Based Engineering (MBE) approaches were adopted rapidly for control software development. Therein, the Unified Modeling Language (UML) (Object Management Group, 2015b) is one of the most wide-spread languages and, hence, the de-facto standard (Muskens et al., 2005) used for software development. Moreover, to support the interdisciplinary system development including, e.g., electrical and mechanical engineering, the Systems Modeling Language (SysML) (Object Management Group, 2015a) is increasingly applied. Using SysML, a shift from a document-centric approach towards an integrated approach using models is envisioned resulting into Model-Based Systems Engineering (MBSE).

Although MBSE approaches support engineers in developing automated production systems from an integrated systems perspective, a multitude of different specific models is still necessary. These models often make use of disparate languages, formalisms, and tools (Broy et al., 2010). Thus, stakeholders create "separate, but interdependent models" (Gausemeier et al., 2009) that *overlap*. They "incorporate elements which refer to common aspects of the system under development" (Spanoudakis and Zisman, 2001). For instance, mechanical engineers focus on the physical layout of an automated production system, whereas electrical engineers address a device-oriented perspective (Feldmann et al., 2012). As a consequence, inconsistencies are likely to occur in between these disparate, heterogeneous models.

In related research, the term *consistency* is often defined using its antonym *inconsistency*: A model (or a set of models) is said to be *inconsistent*, if two assertions in a model (or a set of models) are not jointly satisfiable (Spanoudakis and Zisman, 2001), therefore leading to a situation "in which a set of descriptions does not obey some relationship that should hold between them" (Nuseibeh et al., 2000). If no known contradictions can be identified, the model (or the set of models) is identified to be *consistent*.

Inconsistencies may occur due to manifold reasons. In Lange and Chaudron (2004), the authors investigate the broader concept of *model completeness*. In particular, according to Lange and Chaudron (2004), completeness can be broken down into *well-formedness* of single models (e.g., UML diagrams), *consistency* of overlapping information (e.g., between two UML diagrams), and *diagram completeness* (e.g., missing expected counterparts in a UML diagram such as methods without classes). We refer to such flaws in model completeness as *inconsistencies* in this article. Hence, such inconsistencies can be the result of modellers leaving incompletenesses in their models (Lange and Chaudron, 2004) and of collaborative work between different stakeholders (Hehenberger et al., 2010; Mens et al., 2006).

Inconsistencies can be classified according to a multitude of different characteristics. For the purpose of this article, we mainly distinguish between *intra-model* and *inter-model* inconsistencies. Whereas *intra-model* inconsistencies occur within a single model and result from, e.g., "the imprecise semantics of the UML" (Huzar et al., 2005) in case of software modeling, *inter-model* inconsistencies arise between a set of different models, e.g., in case model artefacts are refined from one model to another. Intra-model inconsistencies are often referred to as well-formedness constraints (e.g., inconsistencies due to objects without names) (Lange and Chaudron, 2004) and are mostly managed by modeling tools (e.g., Eclipse Papyrus Eclipse Foundation, 2015 or No Magic MagicDraw (NoMagic, Inc., 2016) in case of UML and SysML). Inter-model inconsistencies mainly result from consistency or completeness flaws between distinct models (Lange and Chaudron, 2004), e.g., inconsistencies due to wrong or missing links between objects in different diagrams. Although not limited to, we mainly focus on *inter-model* inconsistencies throughout this article.

2.2. Challenges

Due to the manifold inconsistencies in systems engineering, essential challenges need to be tackled by a comprehensive inconsistency management approach.

Challenge 1 (Heterogeneous Engineering Models)**.** As discussed beforehand, during the engineering of automated production systems, a multitude of different stakeholders from different disciplines is involved (Vogel-Heuser et al., 2015). Therefore, engineers use different modeling formalisms, languages and tools (Gausemeier et al., 2009; Broy et al., 2010) as well as different levels of abstraction to express their view on the overall system (Jäger et al., 2012). For instance, UML with its multitude of different diagram types already poses the potential for many inconsistencies (Lange and Chaudron, 2004) – with additional types of models as required for the automated production systems engineering, the likelihood for inconsistencies to occur becomes even higher. The resulting heterogeneity of engineering models poses a major challenge (Gausemeier et al., 2009; Broy et al., 2010) that

must be tackled by a comprehensive inconsistency management approach.

Challenge 2 (Various Inconsistency Types)**.** In addition, a multitude of different types of inconsistencies needs to be incorporated for the automated production systems domain (Feldmann et al., 2015b; Herzig et al., 2011). Respectively, a comprehensive inconsistency management approach must be capable of specifying, diagnosing and handling the *multitude of types of inconsistencies* that can occur during automated production systems' engineering.

Challenge 3 (Lack of Inconsistency Life Cycle Documentation)**.** Important to the automated production systems domain is *documentation*. For instance, the late engineering phases – e.g., testing and commissioning – shortly before system operation are mostly critical for engineering projects as tasks must be performed efficiently and under high time pressure. As a consequence, it is likely that engineers make mistakes or implement "quick hacks", which may lead to *technical debt* (Vogel-Heuser and Rösch, 2015) – namely extra development time and costs that arise when refraining from applying the optimal solution. Therefore, we regard the *lack in documentation of design decisions in case of inconsistencies* as essential challenge to be tackled by an inconsistency management approach.

Challenge 4 (Lack of Comprehensive Tool Support for Managing Inconsistencies)**.** Whereas a variety of (theoretical and practical) inconsistency management frameworks have been proposed in related research work, essential to the application of such frameworks to the automated production systems domain is an appropriate support of stakeholders. Therefore, it is essential that comprehensive tool support is provided that allows these stakeholders to diagnose and handle the inconsistencies that are relevant for them.

3. Application example and preliminaries

To illustrate our inconsistency management framework, this section introduces a running application example as well as the preliminaries of our concept. Within the application example, two fictitious modeling languages are used, namely the *taxonomy modelling language (TML)* and the *hierarchy modelling language (HML)*. Whereas TML uses a tabular notation of component lists, HML corresponds to a SysML-style modeling language to define a system's structure. We chose these two fictitious modeling languages as they are representative for a typical digital transformation process in companies in the automated production systems domain – from less sophisticated management frameworks that are typically implemented by means of tabular sheets (represented through our TML) to more sophisticated modeling frameworks by means of domain-specific languages (such as our HML).

Within our application example, the aim is to specify the logical structure of an automated production system. In industrial applications, companies apply component lists to specify a unique reference to each mechatronic module, machine and plant that are developed and delivered to the customer. Hence, our application example incorporates two distinct models – one modelled in TML and one in HML (see Fig. 1).

To manage such a component list by means of tables, we apply TML as illustrated in Fig. 1a. This figure illustrates a tabular definition of the *Sample Plant* with its according child modules as typically used in reference designation systems such as the IEC Standard 81346-2 (International Electrotechnical Commission, 2009; Göring and Fay, 2012). Accordingly, the tabular *Sample Plant* definition specifies properties such as the modules' names as well as their mass values in *kilograms*. As discussed beforehand, our fictitious company intends to extend the tabular notation in TML towards a model-based management of the component list within HML. Therefore, the HML model in Fig. 1b applies a UML object diagram-style notation. The overlap between both TML and HML models is obvious, as both models specify the *Sample Plant*'s hierarchy in a similar but different manner. However, HML distinguishes between different types of entities – namely *components* and *modules* – and choosing specific mass units such as *lbs*.

During the transformation process from a tabular notation in TML towards an object diagram-style notation in HML, our fictitious company now faces the challenge to ensure that both models are consistent with each other. To automate reasoning tasks such as consistency checks, the models have to be machine-readable and machine-interpretable. As we follow a model-based approach, we make use of metamodeling (Kühne, 2006) as the central technique to define modelling languages and, thus, the structure of their corresponding models. In particular, metamodels specify the so-called *abstract syntax* of modelling languages (i.e., the language concepts and their relationships) that is the centre of the modelling language definition and all other concerns such as the *concrete syntax* (i.e., the graphical model notation) (Brambilla et al., 2012) are based on it. There exist standardized metamodeling languages such as the Meta Object Facility (MOF) (Object Management Group (OMG), 2003) for defining modeling languages that may be seen as pendant to Extended Backus–Naur Form (EBNF) (Wirth, 1977), which is the foundation for specifying textual languages.

MOF is based on a core subset of UML class diagrams and includes classes, attributes, and references. A metamodel gives the intentional description of all possible models within a given language. Using both metamodels for TML and HML as defined in the appendix in Fig. A.1, the means to formalize the constraints that must be obeyed by both languages are provided.

Practically, metamodels are instantiated to produce models, which are in essence object graphs, i.e., they consist of objects (instances of classes) representing the modeling elements, object slots for storing values (instances of attributes), and links between the objects (instances of references), which have to conform to the UML class diagram describing the metamodel. Therefore, models are often represented in terms of UML object diagrams if their concrete syntax is neglected. This is especially true when models are automatically processed by the computer. In Fig. 2 we show the models illustrated beforehand in Fig. 1 in their abstract syntax. As can be seen from the figure, models in abstract syntax cover all the content that is stored in the models in a generic object representation, which is typed by the associated metamodel. By that, our approach is independent from the concrete software tools applied as all models have an associated MOF-based metamodel. This capability is one of the major cornerstones of our inconsistency management approach.

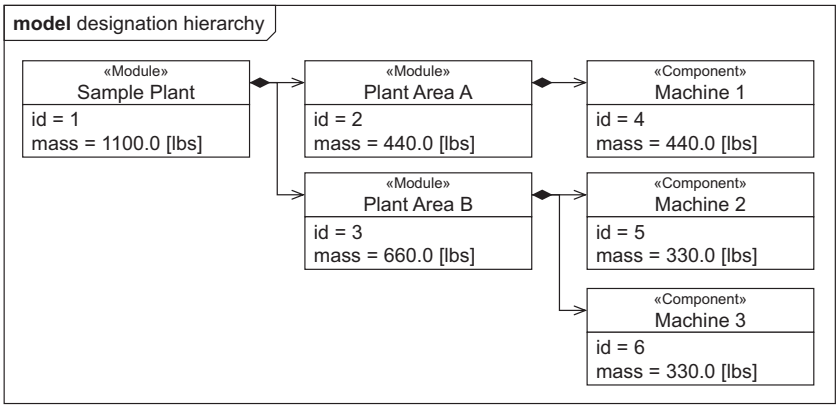
In previous research work (Feldmann et al., 2015a), we observed that inconsistencies such as name mismatches, value- and/or type-related inconsistencies as well as structural inconsistencies are typical inconsistencies for the automated production systems domain. We therefore investigate the following inconsistencies throughout our application example:

1. Each *Taxonomy* in *TML* models should have an equivalent *Hierarchy* in *HML* models. In particular, the *Taxonomy*'s *name* should be consistent with the *Hierarchy*'s *name*.
2. Each *Type* in *TML* models should have an equivalent *Entity* (i.e., a *Module* or *Component*) in *HML* models. In particular, the *Type*'s *name, id* and *massInKg* values should be consistent with the *Entity*'s *name, identifier* and *massProperty* values.
3. The hierarchical arrangement between both the *TML* and *HML* models should be identical. That means, that each *Type*

table designation taxonomy

ID	Name	Mass (kg)	Parent
1	Sample Plant	500.0	
2	Plant Area A	200.0	Sample Plant
3	Plant Area B	350.0	Sample Plant
4	Machine 1	200.0	Plant Area B
5	Machine 2	150.0	Plant Area B
…			

(a) TML example: Tabular component list definition



(b) HML example: SysML-style component list definition

Fig. 1. Models of the application example in concrete syntax.

and its associated *nestedTypes* should have an equivalent *Module* with associated *nestedEntities*.

As can be seen from the exemplary models in Figs. 1 and 2, we injected violations of these consistency rules: First, while *Machine 1* is assigned to *Plant Area B* in the TML model, it is assigned to *Plant Area A* in the HML model. Moreover, the HML model introduces the additional component *Machine 3*, which is not included in the TML model. Finally, as a different unit system is used for the HML model compared to the TML model it must be ensured that the mass values are defined accordingly in both models. We will use these simple inconsistencies as the basis to illustrate our concept in the following sections.

4. Model-driven inconsistency management framework

Within this section, we introduce our inconsistency management framework that aims at the specification, diagnosis and handling of inter-model inconsistencies. We use the term *inter-model inconsistency* in the spirit of Nuseibeh et al. (2000) to denote any situation "in which a set of descriptions [that is, the set of our engineering models] does not obey some relationship that should hold between them". Certainly, it is impossible to conclude whether a set of models is (fully) consistent or not (Herzig et al., 2011). Especially in the domain of automated production systems, engineering is subject to ambiguities and unknown information, which is extended throughout the engineering phases. As a consequence, it is impossible to capture all knowledge required to conclude on whether a set of models in the automated production systems domain is fully consistent or not. The best one can do is to diagnose inconsistencies due to known consistency rules (Herzig et al., 2011), e.g., from design catalogues or engineering guidelines, that describe the conditions for an inconsistency to be present or not. Using these rules, if no conflict according to the specified consistency rules is found, we can at least conclude whether the set of models is free of known inconsistencies.

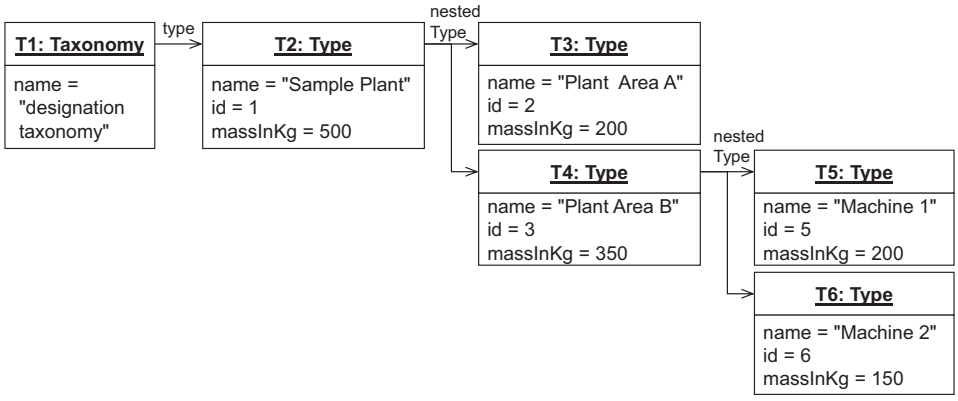
The remainder of this section is structured as follows: We first introduce the main concepts of our framework and define its core components in Section 4.1. Second, in Section 4.2 we discuss the design rationale and the conceptual architecture of our framework. Third, the key components are introduced in the subsequent sections – namely the linking languages (Section 4.3), the pattern-based specification of inter-model inconsistencies (Section 4.4) as well as the inconsistency diagnosis and handling mechanism (Section 4.5).

4.1. Main concepts

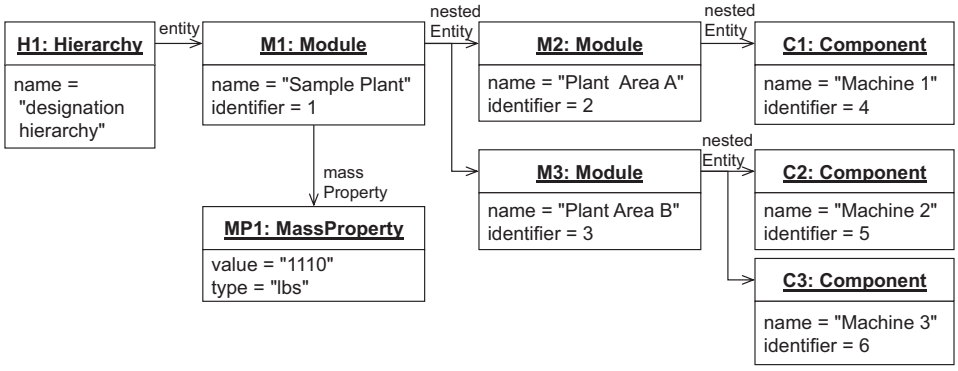
The starting point for our inconsistency management framework is considered to be a set of related engineering models. As introduced in our previous work (Feldmann et al., 2016b), we can define our inconsistency management problem *IMP* as a 5-tuple as shown in (1).

$$IMP = < BM, LM, CR, IDR, IHA > \quad (1)$$

This 5-tuple contains of the following elements:



(a) TML example model in abstract syntax



(b) HML example model in abstract syntax (excerpt, mass properties not visualized for all model elements)

Fig. 2. Models of the application example in abstract syntax.

- *Base models. BM* refers to a set of *base models*, i.e., the different engineering models $BM_1 \dots BM_n$ created during the development process.
- *Link models.* Our *LM* is a set of link models that specify the relationships between the base models, i.e., $LM_{ij}(BM_i, BM_j)$ for $i = 1 \dots n$, $j = 1 \dots n$, and $i \neq j$ as links are usually not defined within a single engineering model.
- *Consistency rules.* Accordingly, *CR* represents a set of consistency rules $CR_1 \dots CR_m$ for the linked base models $< BM_i, LM_{ij}, BM_j >$. BM_i, BM_j will, hence, be said to be inconsistent with respect to CR_k for $k = 1 \dots m$ – given the relationship between them as specified by the link model LM_{ij} – if it can be shown that the interpretation of CR_k by an evaluation function α is not satisfied, i.e., $\alpha(CR_k, < BM_i, LM_{ij}, BM_j >) \rightarrow \bot$.
- *Inconsistency diagnosis results. IDR* is a set of inconsistency diagnosis results produced by α as a side-product for all CR_k evaluations – both satisfied and unsatisfied ones – for the given base models and their link models $< BM_i, LM_{ij}, BM_j >$.
- *Inconsistency handling actions.* Finally, *IHA* is the set of inconsistency handling actions, which transform $BM_1 \dots BM_n$ into $BM_{1'} \dots BM_{n'}$ and $LM_{ij}(BM_i, BM_j)$ into $LM_{ij'}(BM_{i'}, BM_{j'})$ such that the evaluation function α is satisfied for all CR_k, i.e., $\alpha(CR_k, < BM_i, LM_{ij}, BM_j >) \rightarrow \top$. Especially as inconsistencies do not necessarily lead to engineering errors, but rather can also denote situations in which system alternatives still need to be revised and compared, these handling actions not only cover inconsistency resolution, but also tolerating and ignoring actions. Hence, our evaluation function α must also incorporate this kind of information when diagnosing and handling inconsistencies.

As can be seen from this definition of an inconsistency management problem *IMP*, the crucial parts of our framework are the link models *LM* and the consistency rules *CR*. Whereas the representation of involved base models *BM* is inherent as their metamodels are predefined through their language type, respective dedicated languages are needed to define the other parts of our *IMP*. After discussing the design rationale and the conceptual architecture of our inconsistency management problem in the following section, we will discuss these parts in detail.

4.2. Design rationale and architecture

Based on the evaluation of existing inconsistency management approaches in prior work (Feldmann et al., 2015b) and as discussed in Section 8, we argue that a rule-based inconsistency management framework is an appropriate basis for a comprehensive and extensible inconsistency management framework, which can be applied to a broad variety of engineering applications in the automated production systems domain.

To fulfil the challenges discussed in Section 2, a conceptual architecture for our inconsistency management framework is defined in Fig. 3. This architecture builds upon the previously discussed main concept of linked engineering models. To define these link models, linking languages need to be created (see Section 4.3). Pattern-based Modeling Language for Model Transformations (PaMoMo) triple graph patterns specify the consistency rules that need to be fulfilled by link and base models (see Section 4.4). Accordingly, using an inconsistency diagnosis and handling engine, it can be concluded whether the link models conform to these patterns or not (see Section 4.5). The following

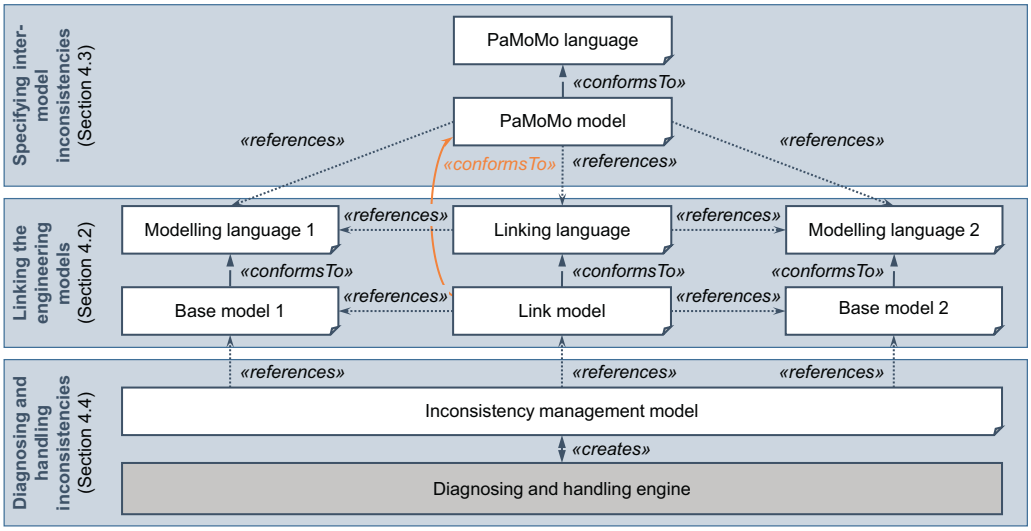


Fig. 3. Overview of the conceptual architecture of the inconsistency management framework.

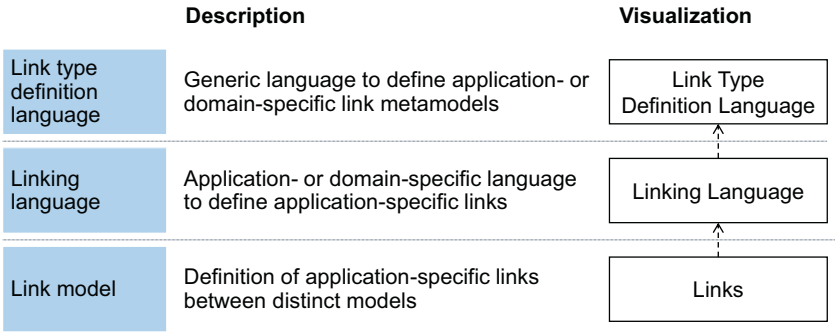


Fig. 4. Overview of the three-layered linking concept.

sections introduce all these conceptual components of our inconsistency management framework.

4.3. Linking the engineering models

Linking between engineering models is an essential part of our inconsistency management framework. Therefore, this section introduces a link model $LM_{ij}(BM_i, BM_j)$ to specify the relations between distinct engineering models. Accordingly, the main content of a link model is a set of links, which relate a set of models on the left-hand side (henceforth called *left models*) with models on the right-hand side (henceforth called *right models*) via proxy elements (referred to as *middle model*) (Bézivin et al., 2006a).

To describe these different links, we envision a three-layered concept[1] as denoted in Fig. 4. The most abstract *link type definition language* therein allows stakeholders to define their application-specific or domain-specific *link metamodels* (i.e., their own link languages) on the middle layer. By means of these *link metamodels*, concrete *link models* (i.e., a set of specific link instances between different models) can be defined within certain applications on the bottom layer. These components are introduced in the following.

[1] The particular challenge to implement a three-layered approach in two-level metamodeling is out of the scope of this article and, hence, not discussed in detail. To overcome this challenge, we have developed a particular model-to-model transformation, which transforms instances of the link type definition language into a metamodel. By this, we follow the metamodel lifting pattern which has been already proposed in previous work to tackle the limitations of two-level metamodeling approaches (Langer et al., 2012; Zschaler et al., 2009; Drivalos et al., 2008).

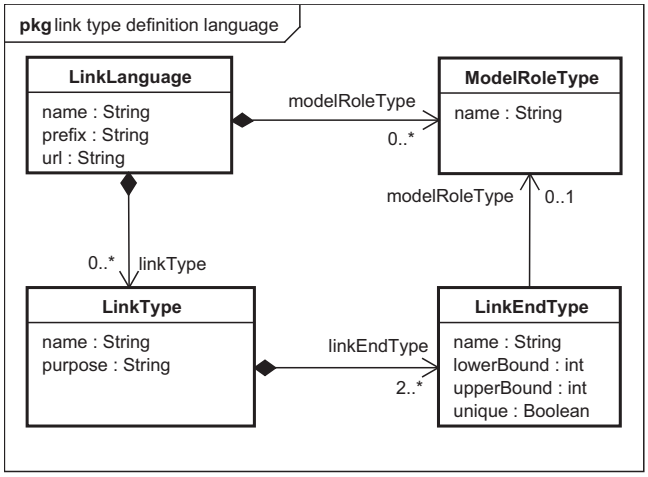


Fig. 5. Language for modeling link types.

4.3.1. Introducing the link type definition language

Different link types are used throughout an inconsistency management problem *IMP*. For instance, modellers use *satisfies*, *refines*, and *is-equivalent-to* links as abstract concepts in the domain of systems engineering. It is, therefore, helpful and useful that stakeholders can pre-define their required link types depending on their domain-specific purposes. Accordingly, we introduce a domain-specific link type definition language (see Fig. 5), which allows to define an application-specific *LinkLanguage* that consists

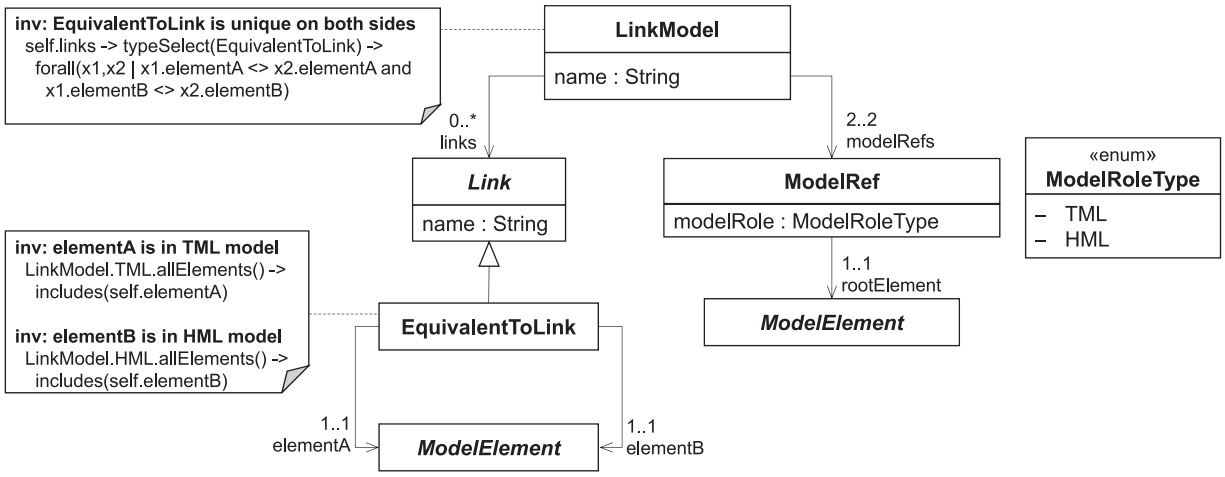


Fig. 6. Equivalence Linking Language required for the application example.

of specific *LinkTypes* for different purposes. By means of the according *LinkEndTypes*, the multiplicity of such links can be defined. Hence, this allows us to handle *n-to-m* links (i.e., *n* model elements in one engineering model that are linked to *m* elements in another engineering model), *1-to-1* links, as well as *1-to-n* links. The link type definition language is based on previous work on trace links, which also allows for specifying certain link type specifics (Drivalos et al., 2008).

Furthermore, the language supports the definition of a specific *unique* property, which defines whether links are allowed to share their linked elements with other links of the same type or not. In particular, for each *LinkEndTypes* of a *Link* this property can be specified.

Finally, we can also align the *LinkEndTypes* with particular *ModelRoleTypes*. This allows us to express whether a certain link type is always used consistently in the specified consistency rules.[2] In more detail, the *ModelRoleType* represents the role a model can take in the engineering process such as a requirement model, a design model, or a validation model. If a *LinkEndType* is pointing to a *ModelRoleType*, it requires the elements, which are linked later on with this *LinkEndType*, to be contained in the model which has the assigned role. For instance, by having this concept in our link type definition language, we can ensure for our application example that a specific link type is always linking between a TML model entity and a HML model entity—but not between elements within one of these models.

4.3.2. Defining specific link metamodels

By means of the link type definition language, in principle all different types of link languages can be defined. An exemplary link language that allows for modeling entity equivalence between our TML and HML models is illustrated in Fig. 6. Using this link language, first, specific links can be expressed to denote any overlap between two or more distinct engineering models. Moreover, consistency rules make use of the types of this link language to specify the relations that must hold between two engineering models in particular situations.

4.3.3. Modeling specific links

By means of the link metamodel, concrete link instances can be modelled. An exemplary, conceptual overview on the links between the models of the application example is illustrated in Fig. 7. As can be seen from the figure, the entities between both the TML and HML model are linked to each other via equivalence links. Therein, links between the *Taxonomy* (TML) and the *Hierarchy* (HML) instances exist. Moreover, equivalences are defined between the *Types* in the TML model and the *Entities* in the HML model. It can be also clearly seen in the figure that the multiplicity and uniqueness constraints of the equivalence links are fulfilled. However, a link that connects *Machine 3* in HML with a respective type in TML does not exist—hence, there is a potential inconsistency. Thus, additional constraints on the links in combination with the respective modeling languages of the linked models are needed, e.g., to guarantee this completeness property that each modeling element is mapped to a corresponding counterpart.

4.4. Specifying inter-model inconsistencies

Consistency rules (*CR*) put the link types into context for a given scenario, thereby specifying the semantic context, in which a specific link type is meant to be used. In principle, any language that produces a logical (*true* or *false*) statement for a given modeling pattern can be used to formulate the required consistency rules *CR*. Our prototypical implementation of the inconsistency management framework makes use of the Epsilon Validation Language (EVL) as discussed in Section 5.

Hence, by means of these *CR*, context-depending constraints can be developed. Such a context can, e.g., represent the used modeling languages, modeling methodologies, as well as company- or project-specific guidelines. Within our concept, we aim at a declarative approach, which simplifies the specification of such rules for stakeholders through a graphical modeling language. Respective mechanisms to evaluate these rules to allow for meaningful interpretation and execution of these rules can be put in place, e.g., to decide whether a link is valid or not (consistency), detect missing links as well as repairing invalid ones (correctness). As a basis to allow for defining such rules, link types must also be associated with the metamodels, i.e., the modeling languages, of the different engineering models.

4.4.1. Using pamomo to graphically model consistency rules

In the following, we discuss our approach at the example of the *equivalence* relations between the TML and the HML models in our application example. In particular, we intend to ensure *equivalence* links between each *Taxonomy* in the TML model as well as each *Hierarchy* in the HML model. Analogously, each *Type* in the TML model should be equivalent to an *Entity* in the HML model. As we do not allow for arbitrary links between elements of the TML and HML models, we have to formulate additional constraints for the link models – for instance, as *Types* in TML that have further children *Types* must be represented as *Modules* in HML.

[2] Consistency of consistency rules is, however, out of the scope of this article.

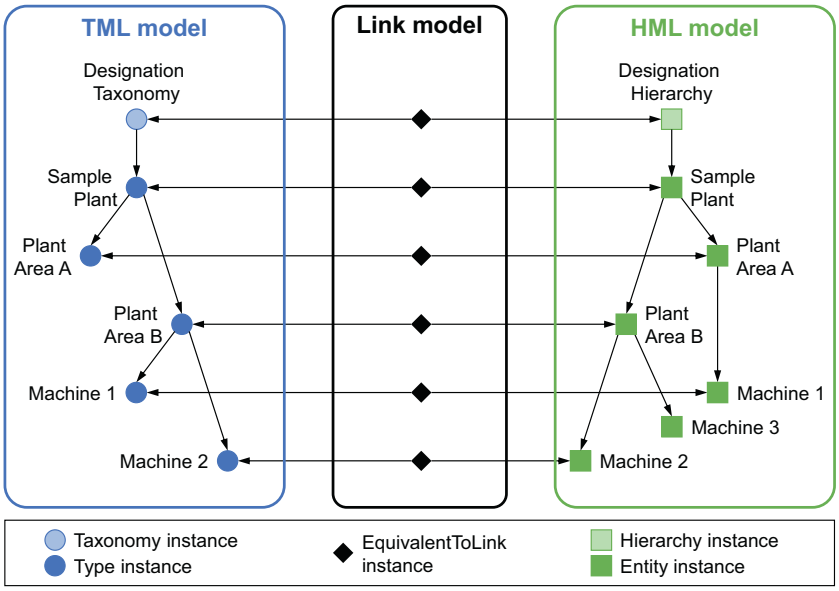
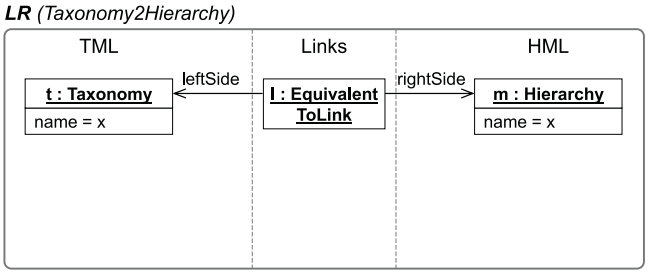
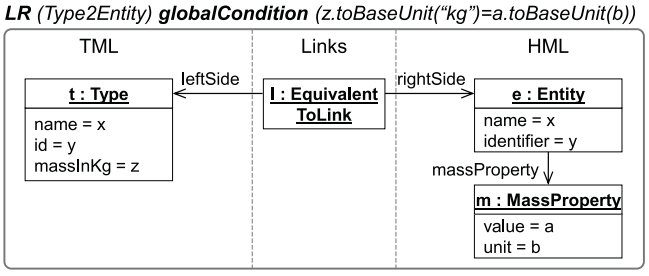


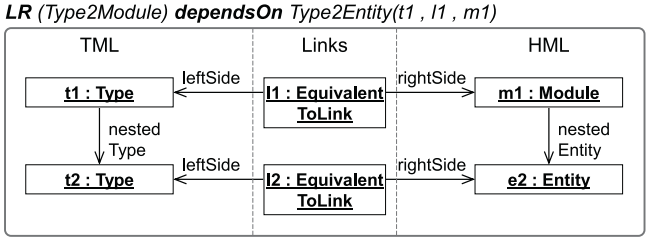
Fig. 7. Conceptual overview on the links between the TML and HML models in the application example.



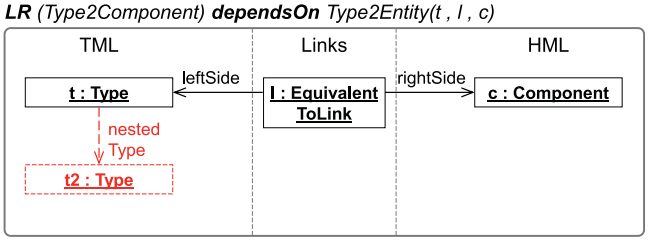
(a) Consistency rule for equivalence between Taxonomy and Hierarchy



(b) Consistency rule for equivalence between Type and Entity



(c) Consistency rule for equivalence between Type and Module



(d) Consistency rule for equivalence between Type and Component

Fig. 8. PaMoMo Triple Graph Patterns (TGPs) for the application example.

Accordingly, these constraints describe the valid link models that should be met by both model types. We define these types of constraints for valid link models as Triple Graph Patterns (TGPs) using PaMoMo (Guerra et al., 2013a, 2013b). These TGPs allow for the description of inter-model constraints by formulating individual graph patterns that are matched against the individual engineering models and relate them with inter-model constraints. One can think of this approach as binding model elements coming from the left models, middle model and right models to different variables and, subsequently, formulating conditions on them.

Consider the PaMoMo TGPs in Fig. 8. These patterns describe the expected relationships between the models of our application example. For instance, for each *Taxonomy* in the TML model, we consider one corresponding *Hierarchy* in the HML model (see Fig. 8a). The correspondence is expressed as a name equivalence for the two given elements. In case the name equivalence is given, an equivalence link has to exist between these two elements to document their relationship. As equivalence links are intended to be bidirectional, each *Taxonomy* needs a corresponding *Hierarchy* and vice versa. Therefore, the TGP provides an additional attribute describing its direction. Three different directions can be defined: L2R (i.e., each element on the left hand side must be mapped an element on the right hand side), R2L (i.e., each element on the right hand side must be mapped to an element on the left hand side) and LR (i.e., there is a one-to-one mapping between elements on the left and right hand side). Therefore, for our equivalence

between *taxonomies* and *hierarchies* the direction of the pattern is marked as LR; hence, the pattern is ensured as a one-to-one mapping between the models.

More formally, given an occurrence o of the right hand side pattern P_r in the right model M_r, an occurrence o of the left hand side pattern P_l in the left model M_l as well as an occurrence o of the middle pattern P_m in the middle model M_m with an according inter-model consistency condition *cond*, which is evaluated to be fulfilled for the bounded elements of $o(P_l)$, $o(P_m)$, $o(P_r)$, we can define the semantics for TGPs directions L2R (2) and R2L (3) as well as for LR (both (2) and (3) must be fulfilled).

$$\forall\, o(P_l) \in M_l \rightarrow \exists\, o(P_r) \in M_r \wedge o(P_m) \in M_m \mid cond(o(P_l), o(P_m), o(P_r)) \quad (2)$$

$$\forall\, o(P_r) \in M_r \rightarrow \exists\, o(P_l) \in M_l \wedge o(P_m) \in M_m \mid cond(o(P_l), o(P_m), o(P_r)) \quad (3)$$

By means of these equations and the given PaMoMo TGPs (see Fig. 8), it is precisely defined, which links must exist between the TML and HML models. Accordingly, more complex patterns can be expressed as needed for the consistency rules regarding the equivalence between *Types* and *Entities* (see Fig. 8b), between *Types* and *Modules* (see Fig. 8c) as well as between *Types* and *Components* (see Fig. 8d).

First, *dependsOn* relations can be defined in order to denote whether a TGP is only applicable in case another TGP is already evaluated to be true. For instance, the *Type2Module* and *Type2Component* constraints are only considered in case the *Type2Entity* constraint is evaluated to be fulfilled. By this, TGPs are more easily definable by reusing certain elements which are already bounded other TGPs. For instance, *Type2Module* and *Type2Component* do not have to take care of evaluating the correspondence for the mass properties as this is already achieved by *Type2Entity*.

In addition to positive conditions, i.e., the presence of elements, negative conditions can be defined to denote the absence of elements in the involved models. For instance, the *Type2Component* constraint ensures that *Types* without nested *Types* in TML are represented as *Components* in HML. This is achieved by stating that there should not be any nested *Type* for the context type.

Finally, so-called *globalConditions* are used to incorporate background knowledge and to express conditions which span over the different compartments of a TGP. As an example, in order to ensure that the mass properties defined in the TML and HML models are equivalent to each other, a respective conversion function *toBaseUnit* is needed for the *Type2Entity* constraint[3]

4.4.2. Combined evaluation of pamomo rules and uniqueness constraints

PaMoMo in combination with the uniqueness constraint of the link type language allows to define different *mapping* properties, which define whether a link is meant to be a general *relation*, a general *function* or more specifically an *injective, surjective* or *bijective* function. We consider the link type as a relation r, which maps elements from its domain, i.e., the left base model BM_i, to elements in its range, i.e., the right base model BM_j. Thus, links may be considered as ordered pairs of elements from BM_i and BM_j. If unrestricted, we allow for any subset of the Cartesian product $BM_i \times BM_j$. This gives us the following definitions for the five mapping kinds we provide for *1-to-1* link types. We discuss it in particular for the example of mapping TML *Types* (as the domain) to HML *Entities* (as the range).

[3] Value mediations to implement the conversions between units are, however, out of the scope of this article.

Relation

If no PaMoMo pattern is specified for a link type, no constraints on the relation are imposed. Thus, as already mentioned, any subset of the Cartesian product $BM_i \times BM_j$ is considered valid. For our application example, this would mean that some types are mapped to some entities, some types and entities may remain unmapped, and some types/entities may be mapped several times to different entities/types, respectively.

Function

The link type is considered to be a function f if $\forall x \in BM_i \exists y \in BM_j | y = f(x)$. This mapping kind is expressible in our approach by defining a PaMoMo pattern for the particular link type as L2R. In our application example, this would mean that every type has to be mapped to exactly one entity, i.e., there is no type in TML which has no correspondence in HML. However, f may be (non-)injective and (non-)surjective in all possible combinations as no stronger constraint is specified for the link type in this case of having a general function.

Injective Function

f is considered to be a function fulfilling $\forall x, x' \in BM_i | x \neq x' \Rightarrow f(x) \neq f(x')$. This mapping kind is expressible in our approach by setting the PaMoMo pattern to L2R, but in addition, setting the link end type on the TML side as *unique*. In our application example, this would mean that every type has to be mapped to its unique entity, i.e., there is no entity in HML which has more than one correspondence in TML but maybe none.

Surjective function

f is considered to be a function fulfilling $\forall y \in BM_j \exists x \in BM_i | y = f(x)$. This mapping kind is expressible in our approach by setting the PaMoMo pattern R2L. In our example, this would mean that every entity has to be mapped to at least one type, i.e., there does not exist an entity in HML which has no correspondence in TML.

Bijective function

f is injective and surjective. This mapping kind is expressible in our approach by setting the PaMoMo pattern LR as well as setting both link end types as *unique*. In our example, this would mean that we have one-to-one correspondences between the model elements of both models. Every type/entity has its unique entity/type, respectively.

As we can see for our example by investigating Fig. 6 in combination with Fig. 8, we require a bijective function for the *EquivalentToLink* in combination with the defined PaMoMo patterns as all patterns are defined as LR and for the *EquivalentToLink*, both link end types are unique.

4.5. Diagnosing and handling inter-model inconsistencies

Given that the base models *BM*, link models *LM*, and the according inter-model consistency rules *CR* are available, respective mechanisms to reason on the consistency and completeness of the link models can be put in place. In order to make the according inconsistency management process flexible and interactive, we provide an explicit representation of the diagnosis and handling of inconsistencies. The respective building blocks for such an interactive inconsistency management procedure are introduced in the following.

4.5.1. Reason on specified consistency rules

Four distinct cases can be identified to ensure both correctness and completeness of the respective relationships between the involved engineering models. These different cases lead to disparate

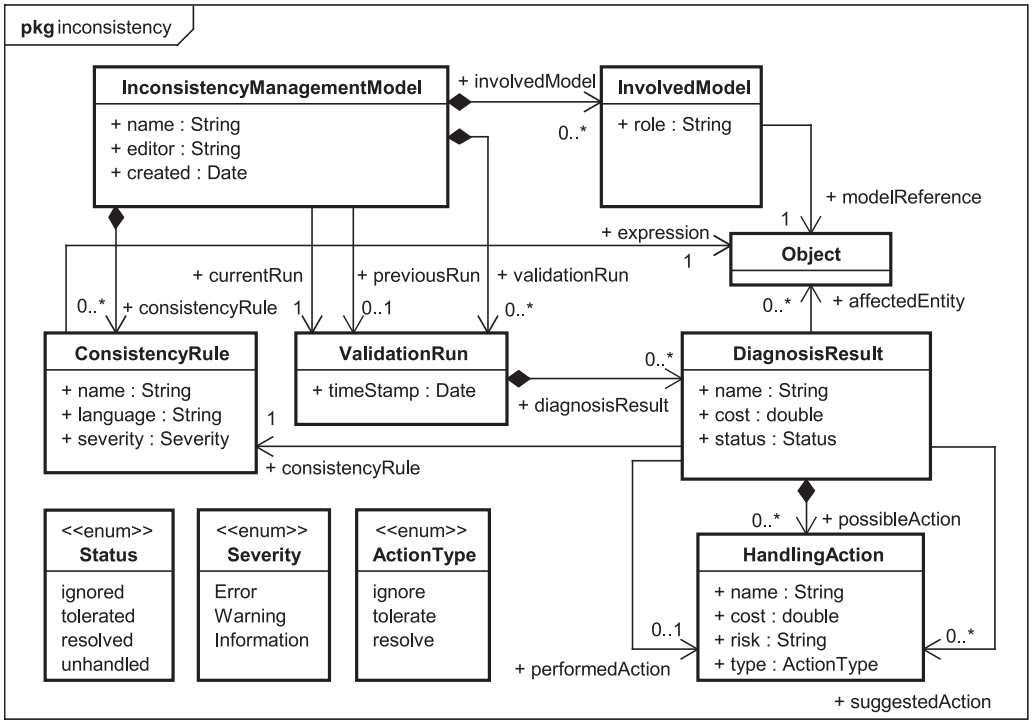


Fig. 9. Relevant excerpt of the inconsistency management metamodel (extended from Feldmann et al. (2016b)).

diagnosis results *IDR* and, hence, to different actions *IHA* that need to be taken upon diagnosing an inconsistency:

1. *Link model generation*: In case the engineering models are unlinked, i.e., no link model is available between two models, new links must be generated. Hence, the goal is to generate a link model based on a given set of consistency rules.
2. *Link model validation*: In case the engineering models are already linked, it must be validated whether the existing models fulfil the given set of consistency rules.
3. *Link model improvement*: In case an initial link model does not obey all given consistency rules, the goal is to improve the existing link model based on the given set of consistency rules.
4. *Link model maintenance*: In case an initial link model does exist and the base models evolve due to necessary changes, the goal is to adapt the existing link model based on the given set of consistency rules in order to reflect the changes in the base models.

As the TGPs are declarative specifications with precise semantics, all four mentioned cases are supported in our approach.

4.5.2. Explicit diagnosis and handling of inconsistencies

In our framework, each time an inter-model inconsistency is found by checking the TGPs, an inconsistency diagnosis result is produced. Therefore, we need support in representing so-called *validation runs* (i.e., points in time, in which the consistency rules are evaluated) to keep the history of inconsistencies during different validation runs. Furthermore, engineers should be supported in adding proposals or meta-information on how the diagnosed inconsistency may be handled (e.g., tolerated, ignored or resolved) later on.

To support these inconsistency handling actions, we provide a dedicated metamodel (cf. Fig. 9) to represent inconsistency diagnosis results by locating, identifying, and classifying the detected inconsistency based on the capabilities provided by the TGPs. In addition, we represent the handling of inconsistencies by documenting different actions such as resolving, ignoring or tolerating inconsistencies. A *resolution* action aims at resolving the inconsistency. In case a user ignores an inconsistency (i.e., the user decides that an inconsistency is no longer of interest), this diagnosis result is not considered in future diagnosis runs. Analogously, the user may decide to tolerate an inconsistency. According to Dávid et al. (2016), such tolerance can, e.g., occur due to *parameter* tolerance (e.g., the user decides to tolerate the inconsistency as parameter values slightly deviate) or due to *temporal* tolerance (e.g., the user decides to tolerate the inconsistency for a certain amount of time). Thus, to allow for such *ignoring* or *tolerating* actions, the consistency diagnosis needs to incorporate previous validation runs when reporting inconsistencies. For instance, if a user diagnoses an inconsistency and decides to ignore that inconsistency during a validation run, we document this decision and, in following validation runs, hide the diagnosed inconsistency. Analogously, if the user decides to temporally tolerate an inconsistency, the diagnosed inconsistency is hidden for the specified amount of time and, if the time is elapsed, included in the displayed results.

Using this interactive approach, the user can decide upon how to handle (i.e., tolerate, ignore, or resolve) an inconsistency depending on the current context. Certainly, in case many inconsistencies occur in the modeling project, the user must take responsibility of a multitude of possible handling actions. In many cases, our automatically generated handling actions can be used to automatically resolve simple inconsistencies (e.g., in case of naming or value conflicts between linked entities). For such cases, the user only needs to decide, which of the linked entities is the “consistent one”. Nevertheless, more complex inconsistencies that cannot be resolved automatically need respective user inputs. Hence, in such cases the user needs to find the appropriate trade-off between (*i*) a highly inconsistent set of models or (*ii*) the effort to handle all the diagnosed inconsistencies.

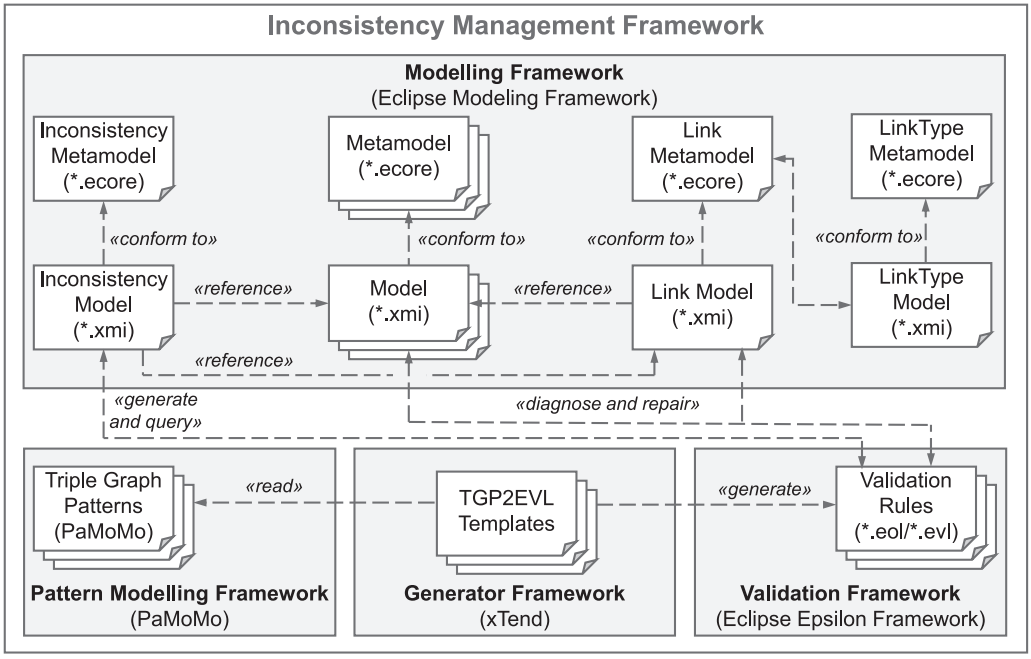


Fig. 10. Overview of the architecture for our prototypical inconsistency management framework (adapted from Feldmann et al. (2016b)).

5. Prototypical implementation

For the purpose of validating our approach, a prototypical implementation was developed, which is illustrated in Fig. 10. Therein, we make use of the Eclipse Modeling Framework[4] for the purpose of defining the metamodels used within our evaluation and modeling the respective model instances. The triple graph patterns as described in Section 4.4 were modelled by means of Pattern-based Modeling Language for Model Transformations (PaMoMo) (Guerra et al., 2013b). The validation technical infrastructure is realized by means of the Eclipse Epsilon Framework[5], which provides the means to query and handle model instances by means of a model-oriented scripting language (Epsilon Object Language, EOL) as well as to formulate and validate intra-model and inter-model consistency rules (Epsilon Validation Language, EVL). For automatically generating the EVL code needed for our evaluation, we developed a model-to-code transformation which is able to compile PaMoMo patterns into EVL constraints. This transformation is realized with Xtend[6], which provides a template-based code generation facility.

For each PaMoMo pattern, the transformation produces corresponding EVL constraints. An excerpt of an example pattern output for the pattern introduced in Fig. 8b is shown in the appendix in Listing B.1. The basic EVL mechanism we are making use of for implementing the patterns is to open up a meta-class and add additional constraints. In particular, we produce constraints which consist of *check, message*, and *fix* blocks. The *check* block contains the core of the transformation by translating the graph patterns provided by PaMoMo to imperative EVL code for querying the model elements needed to reason on whether the stated constraints are fulfilled or not. The *message* and *fix* blocks interact with the inconsistency management model for reasoning on previous validation runs and management of the inconsistencies to find out whether an inconsistency is newly detected or whether it has already been detected or managed in previous runs. Please note that for L2R and R2L patterns, one such EVL constraint is sufficient to implement the pattern, whereas for LR patterns two constraints are needed. Thus, for the given example pattern of Fig. 8b, an analogous EVL constraint is generated for the *Entity* meta-class, which is not shown for the sake of brevity.

In the example illustrated in Listing B.1, a constraint is specified for checking whether for all *Types* in the TML model, a respective *Entity* in the HML model exists that is properly linked and fulfils certain property value correspondences. If a respective inconsistency is identified (i.e., the OCL constraint defined in the *check* part is violated), a diagnosis result is initialized and documented. Within the *fix* part, several alternative fixes are specified. In the excerpt, the shown alternative allows to tolerate the inconsistency; hence, a new *tolerate* handling action is initialized and added to the diagnosis result. In case a subsequent validation run is executed, the diagnosis results are only presented to the user if the previous validation run does not specify the inconsistency to be tolerated (cf. *guard* part of the constraint). Hence, using this implementation, users can easily manage the diagnosed inconsistencies, decide for handling actions to be taken and document their decisions. By means of our model-to-text transformation, all the inconsistency rules expressed in EVL for our application example can be generated from the PaMoMo patterns. The implementation of these constraints is, due to spatial restrictions, not shown in this paper, but can be obtained online.[7]

The user interface for interacting with our inconsistency management approach is shown in Fig. 11. In the upper part, the two models of our application example and the link model, which defines the correspondences between the elements of these two models, are shown. In the middle of this figure, the inconsistency model is shown, which details the evolution of the consistency management process for these two models. In the bottom part of the figure, the current result of running the EVL constraints on the two example models, link model, and inconsistency model are shown. As can be seen, for the reported inconsistencies, actions can be directly selected in order to denote how to deal with

[4] http://www.eclipse.org/modeling/emf/, retrieved March 25, 2019.
[5] http://www.eclipse.org/epsilon/, retrieved March 25, 2019.
[6] http://www.eclipse.org/xtend/, retrieved March 25, 2019.
[7] https://mediatum.ub.tum.de/1356820, retrieved March 25, 2019.

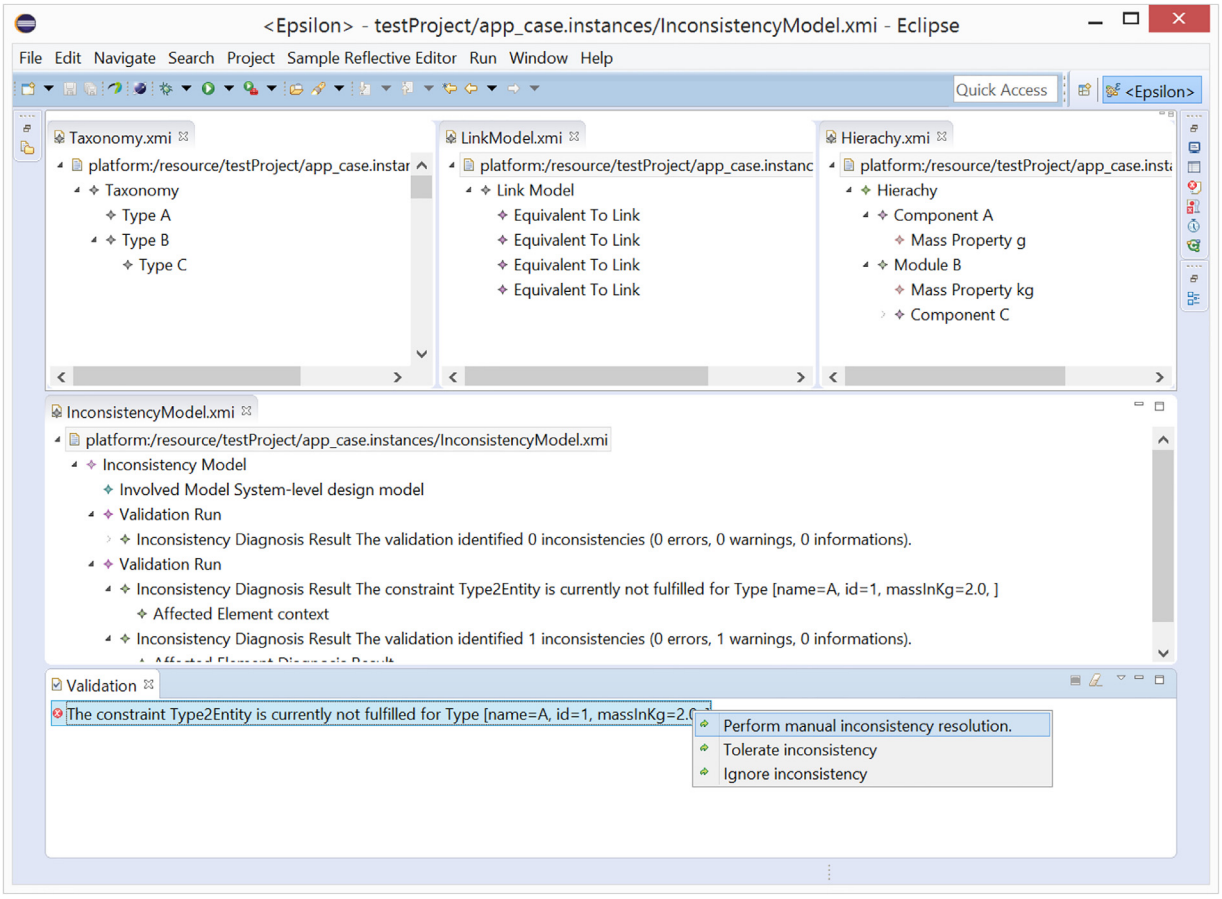


Fig. 11. User interface of the proposed inconsistency management approach.

the inconsistencies. Selecting one of the actions results in the update of the example models, link models, and inconsistency model. In particular, the latter stores how the detected inconsistency has been managed. This information is also used for future validation runs, e.g., to decide if a detected inconsistency is already tolerated/ignored or not.

6. Case study-based evaluation

In this section, we evaluate our inconsistency management framework by means of two distinct model sets. First, we introduce the research questions to be answered by our evaluation in Section 6.1. Second, we introduce the evaluation setup in Section 6.2. Third, the results of our evaluation are discussed in Section 6.3. Finally, we conclude our findings with a discussion of the threats to validity of our results in Section 6.4.

6.1. Research questions

The purpose of this evaluation is to validate (*i*) the *expressiveness* of our approach for inconsistency management, (*ii*) the *efficiency* of the model-based framework compared to manually developing the framework on the code level, and (*iii*) the *scalability* of the approach with an increasing size of models and number of inconsistencies. Therefore, we aim to find responses to the following research questions.

Research Question 1–Expressiveness. Is the proposed approach adequately expressive to support the inconsistency management for typical automated production system engineering projects?

Research Question 2–Efficiency. How efficient is the application of the model-driven framework compared to manually developing the inconsistency framework on the code level?

Research Question 3–Scalability. Is the approach scalable with increasing size of models and number of inconsistencies?

As a basis to perform our evaluation, we apply the prototypical framework as presented in Section 5.

6.2. Evaluation design and setup

In the following, we discuss our evaluation design as well as the setup of our experiments. First, based on the research questions introduced beforehand, we introduce the requirements which our evaluation case must fulfil. Second, we introduce the demonstration case that serves as the basis for our evaluation. Third, the evaluation metrics as well as experimental setup are discussed.

6.2.1. Requirements

To find responses to the previously discussed research questions, our evaluation case must fulfil several requirements. In particular, to provide appropriate estimates of the expressiveness of our inconsistency management framework (see Research Question 1), we need to apply our approach to models and inconsistencies that are representative for engineering processes in the automated production systems domain. Therefore, we use a set of models that covers different engineering phases such as requirement, design and validation within disparate structure and behaviour models. By means of these models we investigate whether

our inconsistency management framework is capable of specifying, diagnosing and handling the representative inconsistencies. In addition, to indicate whether our approach is efficient with regard to its application to automated production systems engineering (see Research Question 2), we need to compare the required specification effort (i.e., effort to model the required PaMoMo constraints) with the hypothetical effort to implement the inconsistency management framework (i.e., effort to develop the consistency rules on a code level). Finally, as a basis to estimate whether our approach scales appropriately with an increasing number of modelled entities and specified inconsistencies (see Research Question 3), our aim is to measure (*i*) the execution time to compile the required EVL constraints from the specified PaMoMo patterns, and (*ii*) the execution time to run the inconsistency diagnosis.

6.2.2. Selected demonstration case

To fulfil these requirements, a demonstration case forms the basis for the evaluation of our inconsistency management framework. This demonstration case is drawn from the so-called Pick and Place Unit (Vogel-Heuser et al., 2014) – a lab-scale demonstrator being developed and maintained at the Institute of Automation and Information Systems (AIS), Technical University of Munich. Based on the demonstration case, we derive two distinct model sets used throughout our evaluation to find initial responses to Research Questions 1, 2, 3. All artefacts of our evaluation can be found online.[7]

Overview of the Pick and Place Unit (PPU)

The PPU is a bench-scale, academic demonstration case that is derived from industrial use cases to evaluate research results at different stages of the engineering process in the automated production systems domain. Although the unit is a simplified case, it is complex enough to demonstrate an excerpt of the challenges that arise during engineering of automated production systems. Therefore, it serves as the case (Runeson and Höst, 2009) for evaluating our inconsistency management framework.

The components of the PPU are industrial components. In addition, the models and scenarios that are developed and continuously evolved for the PPU are derived from real industrial use cases. In particular, as part of the PPU documentation, which is openly available to the research community, 13 distinct basic engineering scenarios including different versions and variants of the PPU's components are defined and maintained within the PPU community. These basic scenarios involve both sequential evolution scenarios, in which components are added, replace and/or updated (such as adding additional handling modules), as well as parallel evolution scenarios, in which different variants of the PPU are engineered (such as using disparate measurement or sensing technologies). All of these scenarios are carefully defined in discussion with industry experts to reflect typical, realistic engineering problems in the automated production systems domain. Therefore, using the PPU, a multitude of distinct models can be created to reflect portions of industrial inconsistency management problems.

Throughout the course of this evaluation, we use the PPU demonstration case to create two distinct model sets. In model set 1, we use three distinct models from the design phase of the PPU, which are based on the pre-work introduced in Feldmann et al. (2016b). Model set 2 addresses the process performance engineering perspective and applies the modeling methodology from Berardinelli et al. (2016) to create the respective models.

Model Set 1: Heterogeneous design models of a lab-scale automated production system

In our model set 1, three distinct models are used, which represent different stakeholders' views during engineering (see Fig. 12). Within an initial *Planning Model*, requirements engineers define the *Requirements* to be fulfilled from an abstract perspective and – based on the *Functions* and *Technologies* demanded – the requirements engineers select the logical *Modules* and *Components* for the PPU. This *Planning Model* is subsequently *refined* by a *SysML Model*, which is used by systems engineers. Therein, the systems engineers describe the logical architecture of the PPU. Following the engineering process, these *Modules* are refined from the ones defined in the *Planning Model* by means of *Blocks* in the *SysML Model*. By that, a systems engineer is able to define the system's architecture as well as the properties and compositions for the system. Finally, for the purpose of estimating whether the requirements imposed in the *Planning Model* are *satisfied*, a *MATLAB/Simulink Model* is introduced. By means of the specified *current velocity* that can be assumed for the *PPU*, the respective *throughput* of work pieces is simulated. This estimation serves as the basis to identify whether the system's design is able to *satisfy* the requirements defined in the *Planning Model*.

Therefore, as can be seen from Fig. 12, in model set 1, we address three distinct link types:

1. *Refines*-links (see ① in Fig. 12) are used to denote the refinement between Modules that have been selected in the Planning model and Blocks in the SysML model. Therein, we require that each refining Block in the SysML model has to be equivalently named to the refined Module in the Planning model. Accordingly, we want the resulting Block hierarchy in the SysML model to be equivalent to the Module hierarchy in the Planning model.
2. *EquivalentTo*-links (see ② in Fig. 12) describe the equivalence between top-level Constants in the Simulink model and Properties in the SysML model. Therein, we assume that each Constant that represents an input of a simulation block in Simulink must be derived from an actual value that is introduced in the SysML model.
3. *Satisfies*-links (see ③ in Fig. 12) are used for denoting that a top-level Display in the Simulink model satisfies a demanded Property in the Planning model. Here, we assume that each Display represents an output of a simulation block in Simulink and must, hence, satisfy a value demanded in the Planning model.

We use these links as a basis to formulate our link metamodel and the consistency rules that must be obeyed by our models.

The relevant excerpts of the metamodels for our model set as well as their intended links are shown in the appendix in Fig. C.1. The *Planning Model*'s metamodel was created from previous work of Kammerl et al. (2015), which extended the so-called *Function–Behavior–Structure* approach (Gero, 1990) for the purpose of modeling technology candidates as well as their relations to respective components. The relevant excerpt of the *SysML* metamodel[8] is based on the Object Management Group's SysML standard (Object Management Group, 2015a). The *MATLAB/Simulink* metamodel was extended from the one provided within the Eclipse Lyo Framework.[9]

Model Set 2: Production process performance evaluation

Our model set 2 investigates the Pick and Place Unit (PPU) (Vogel-Heuser et al., 2014) from a process performance engineering perspective. In particular, this model set is concerned with the evaluation of the performance of a production process executed on the PPU, e.g., cycle response time, utilization, and throughput. In order to do so, a virtual prototype of the reference system is

[8] Note that, for simplification purposes, the SysML profile is illustrated as an extension to the UML metamodel.

[9] http://www.eclipse.org/lyo/, retrieved March 25, 2019.

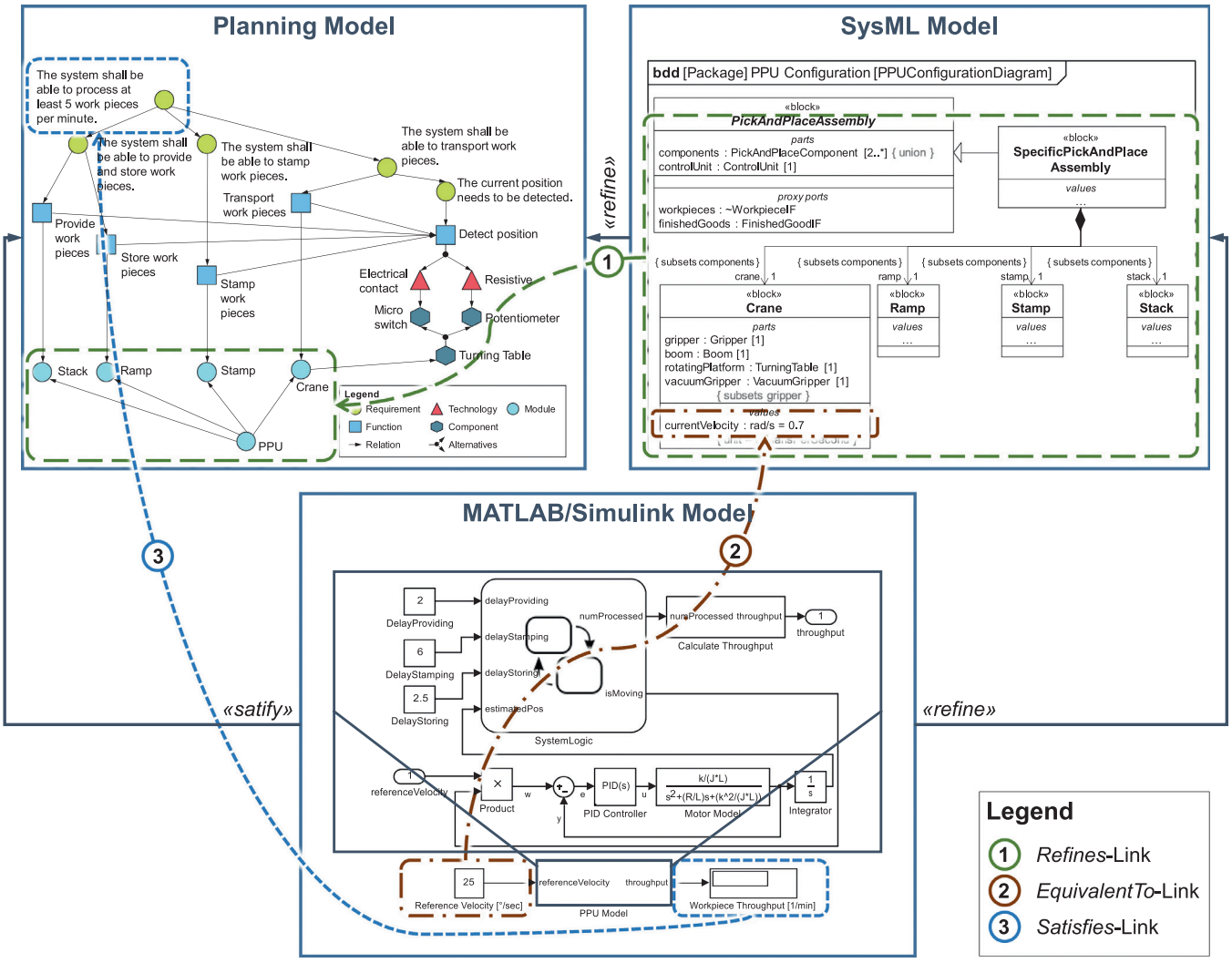


Fig. 12. Overview of the engineering models in model set 1 (taken from Feldmann et al., 2016b).

defined comprising the requirements to be fulfilled as well as the description of the production process, i.e., the different steps needed to process items. For requirement evaluation purposes, a Queuing Network (QN) performance model is established in addition to the production process model to validate the performance of the virtual prototype.

In the given model set, we use three distinct models (see Fig. 13). We focus on one particular performance requirement, which is the cycle time of the system, i.e., the amount of time it takes to produce one item. Within an initial *Requirements Diagram* defined in SysML, requirements engineers define the *Requirements* to be fulfilled from a performance point of view. To fulfil the requirements, a SysML *Activity Diagram* is created by process engineers. In particular, the process engineers describe the production process which should be performed by the PPU. The resulting Activity Diagram has to satisfy the stated performance requirements which are considered as non-functional requirements (NFRs) stated in the *Requirements Diagram*. Finally, for the purpose of *verifying* whether the requirements imposed in the *Requirements Diagram* are actually *satisfied* by the production process expressed by the Activity Diagram, a *Queuing Network* is derived. The Queuing Network has to *refine* the Activity Diagram, e.g., by explicating the expected processing times of the different activities in order to be able to perform the computation of the response time. This estimation serves as the basis to decide whether the process is able to *satisfy* the requirements defined.

As can be seen from Fig. 13, in model set 2, we address three distinct link types to integrate the different models as well as to define and reason about the consistency. While we simply reuse the *Satisfies-Link* (see ① in Fig. 13) and the *Refines-Link* (see ② in Fig. 13) from model set 1, we introduce an additional type *Verifies-Link* to describe that a particular model is used to verify a particular property, which has to be realized by a particular model element.

The relevant excerpts of the metamodels for the second model set as well as their intended links are shown in the appendix in Fig. D.1. The *Requirement Modeling Language* (RML) metamodel[10] was extracted from the SysML standard (Object Management Group, 2015a) as well as profiles targeting NFRs (Gnaho et al., 2013). The relevant excerpt of the *Activity Diagram* metamodel is based on the UML standard (Object Management Group, 2015b). The *Queuing Network* (QN) metamodel, which is used to represent and exchange queuing networks, was reused from Troya and Vallecillo (2014).

[10] Note that, for simplification purposes, the SysML profile is illustrated as an extension to the UML metamodel as done for model set 1.

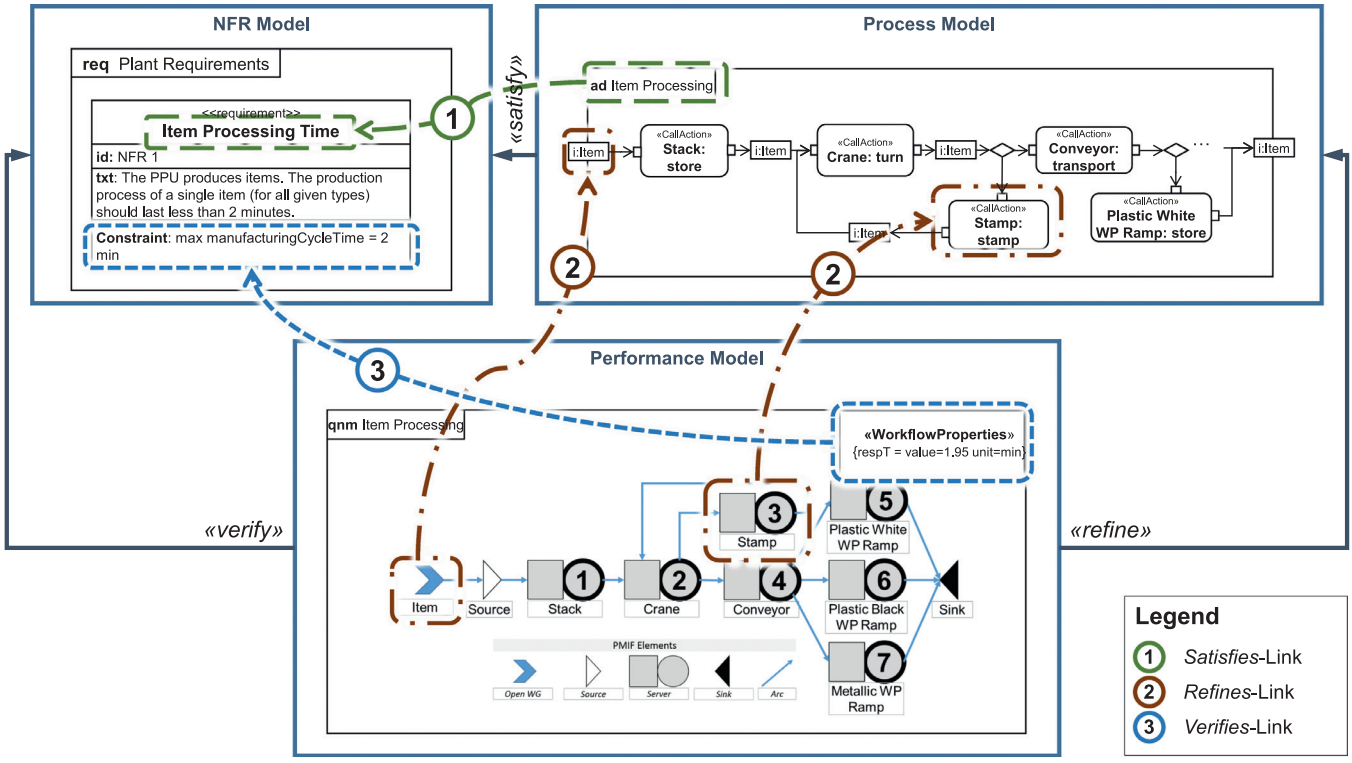


Fig. 13. Overview of the engineering models in model set 2 (based on Berardinelli et al. (2016)).

6.2.3. Metrics and experiment setup

To answer the research questions introduced in Section 6.1, we make use of several strategies. Research Question 1 is answered by analyzing and discussing how the presented approach is applied to define important consistency constraints between heterogeneous artefacts as well as how these constraints are used during the engineering phase. By comparing the complexity of the resulting PaMoMo patterns with their generated EVL counter-parts on the code level for both model sets, we provide indications for the efficiency of our approach (Research Question 2) based on certain model and code metrics. To find indications for the scalability of our framework (Research Question 3), we measure how the execution time (*i*) to generate EVL code from PaMoMo patterns and (*ii*) to diagnose inconsistencies and generate handling actions performs with an increasing number of model artefacts and consistency constraints.

To provide a realistic setup, all our experiments are performed on a standard office PC (Intel(R) Core(TM) i7-5500U CPU @ 2.4 GHz 2.4 GHz, with 24 GB of physical memory on a Windows 7 Professional 64 bit operating system) with the Java Virtual Machine in its version 1.8.0. In addition, we illustrate the experiment results for our model sets 1 and 2 as well as for our application example (see Section 3).

6.3. Results

In this subsection, we present the evaluation results. In particular, we discuss the results for the dimensions *expressiveness* (see Section 6.3.1), *efficiency* (see Section 6.3.2), and *scalability* (see Section 6.3.3).

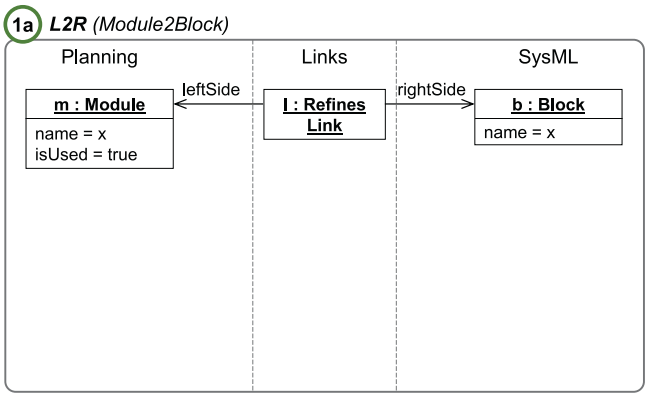
6.3.1. Expressiveness: link metamodels and consistency constraints

We now discuss the *expressiveness* of our inconsistency management framework for the automated production systems domain (see Research Question 1). First, we illustrate how our framework is applied to manage the inconsistency for the given case by defining the required link types for the model sets. From these findings, we subsequently discuss the pattern-based language expressiveness to formulate consistency rules.

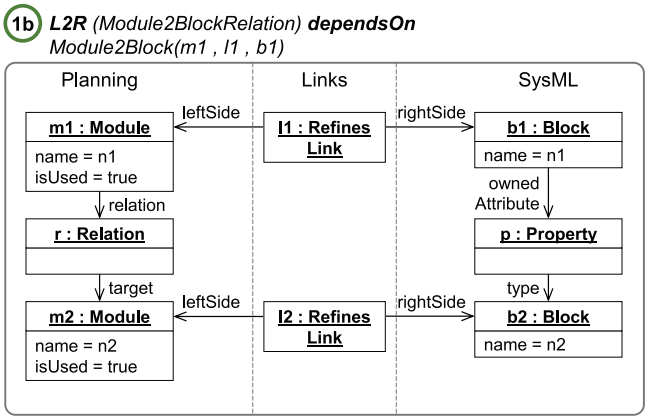
Model set 1

As a basis to formulate our inconsistency management problem for the specific model set 1, a dedicated linking language is defined in Fig. C.2. Therein, the three link types *EquivalentToLink, RefinesLink* and *SatisfiesLink* are defined to cover the three distinct possible links. As all of these links are intended to be used as 1:1-mappings – i.e., a single model element in one model is linked to a single model element in another model – the multiplicity of all link properties – i.e., *leftSide* to denote the source of the link and *rightSide* to denote the target of the link – is defined accordingly. Therefore, by means of the link types introduced in our linking language for model set 1, all possible links that can occur in our model set can be defined.

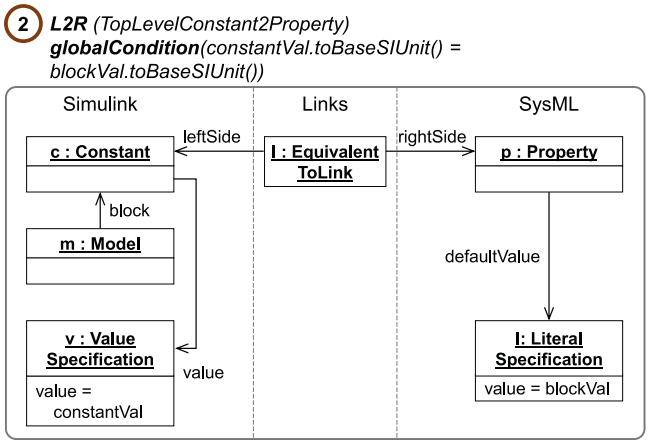
In our model set, consistency rules are formulated for the three link types. In particular, our consistency rules must cover the criteria formulated beforehand – that is, all necessary and sufficient conditions that must be met by the *RefinesLinks, EquivalentToLinks* and *SatisfiesLinks*. To define the different consistency rules, we use PaMoMo patterns as introduced in Section 4.4. Therefore, four distinct patterns are defined in Fig. 14. Within the *Module2Block* pattern (see ⓘa, Fig. 14a), it is demanded that each Module in the Planning Model that is selected to be used for the engineering solution must be linked to an equivalently named Block in the SysML model. As we demand each Module to be linked to a Block, but not vice versa, the pattern is defined as a L2R pattern. Accordingly, a second pattern *Module2BlockRelation* (see ⓘb, Fig. 14b) depends on the *Module2Block* pattern – that is, it is only evaluated if the *Module2Block* pattern is fulfilled for a given set of model elements. This additional pattern then ensures that the hierarchy of Modules in the Planning model is respectively represented by a hierarchy of



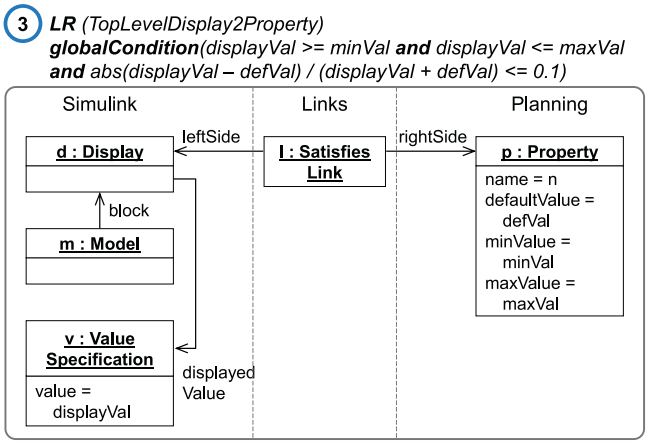
(a) Consistency rule for refinement between a module (Planning model) and a block (SysML model)



(b) Consistency rule for refinement between a module hierarchy (Planning model) and a block hierarchy (SysML model)



(c) Consistency rule for equivalence between top-level constants (Simulink model) and properties (SysML model)



(d) Consistency rule for satisfaction between top-level displays (Simulink model) and properties (Planning model)

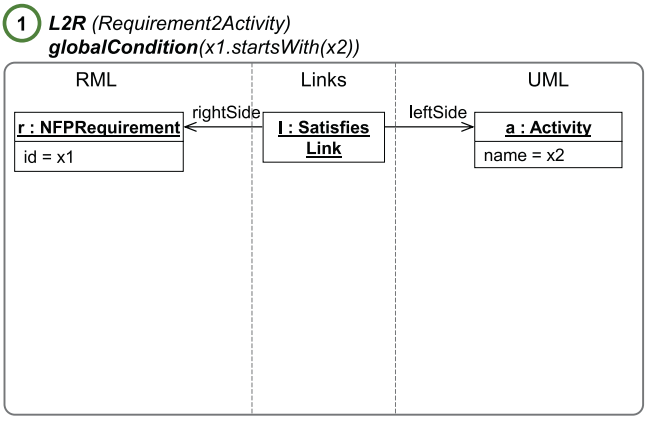
Fig. 14. PaMoMo Triple Graph Patterns (TGPs) for model set 1.

blocks in the SysML model. The pattern *TopLevelConstant2Property* (see ②, Fig. 14c) ensures that each top level Simulink Constant is linked to a SysML property. Therein, it is demanded that the values of both the Constant and the Property are consistent. This is expressed by a *global condition* which is introduced to ensure that, considering their base units, the values are identical. On the contrary, in the *TopLevelDisplay2Property* pattern (see ③, Fig. 14d), no exact equivalence is demanded between Simulink Display and Planning Property values, but it is rather demanded that the ranges are consistent to each other, i.e., that the minimum and maximum values are not violated and that the deviation between the default values is less than 10%.

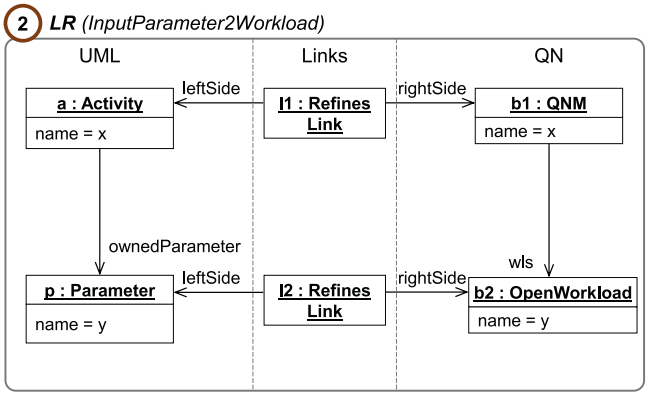
Model set 2

As a basis to formulate our inconsistency management problem for the specific model set 2, a dedicated linking language is defined in Fig. D.2. Therein, the three link types *VerifiesLink*, *RefinesLink* and *SatisfiesLink* are defined to cover the three distinct possible links. While, the *RefinesLink* as well as the *SatisfiesLink* are reused from model set 1, the *VerifiesLink* is specifically introduced for this model set.

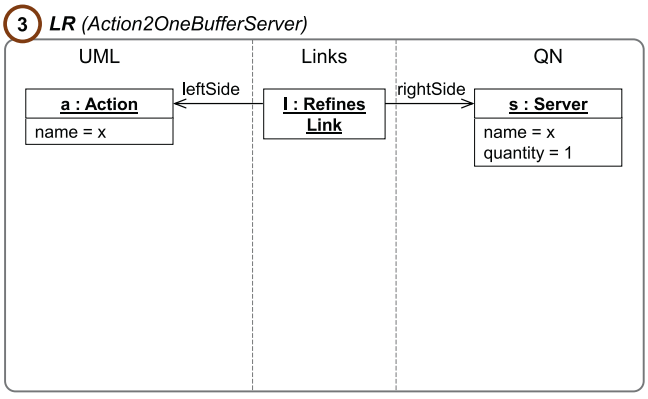
Accordingly, consistency rules for model set 2 are formulated as PaMoMo patterns (see Fig. 15) to ensure the consistency between the process model and the performance model as well as to reason about the relationships between requirements, realizations, and validations. Within the *Requirement2Activity* pattern (see ①, Fig. 15a), it is stated that for each *NFPRequirement*, an *Activity* has to exist which satisfies the requirement. While it still has to be verified that the activity satisfies the requirement, the general relationship between requirements and activities representing production processes are defined. As we demand each *NFPRequirement* to be linked to an *Activity*, but not vice versa, the pattern is defined as a L2R pattern. A second pattern *InputParameter2Workload* (see ②, Fig. 15b) ensures that the *Parameters* of an *Activity* are represented as a *Workload* with the same name in the queuing network. The pattern is defined as LR as the stated relationship has to hold in both directions. This is because, on this abstraction level, changes to an *Activity* entity should be reflected within a *Workload* entity and vice versa. The LR pattern *Action2OneBufferServer* (see ③, Fig. 15c) ensures that each action is linked to a server having exactly one buffer slot. Therein, it is demanded that the names of the *Action* and the *Server* are equivalent. Finally, in the *WorkloadProperty2NFPRequirement* pattern (see ④, Fig. 15d), it is checked if the stated *NFPRequirement* is actually satisfied by the process. Thus, the *ResponseTime* property of the queuing network model has to contain a *value* smaller or equal to the stated *value* of the *NFPRequire-*



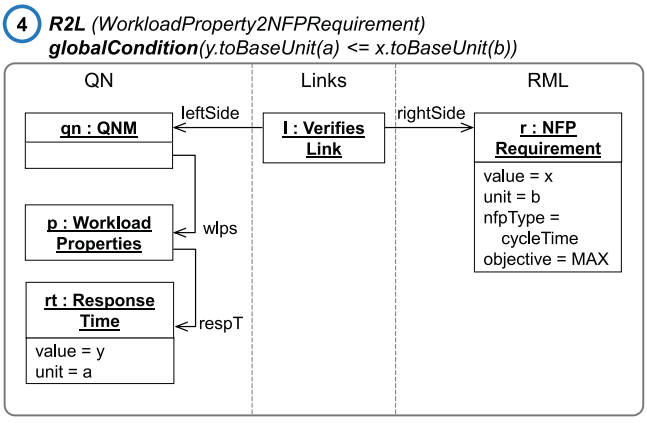
(a) Consistency rule for tracing requirements to activities



(b) Consistency rule for tracing activity parameters to QN workloads



(c) Consistency rule for tracing actions to one buffered servers



(d) Consistency rule for verifying requirements based on QN computation outcome

Fig. 15. PaMoMo Triple Graph Patterns (TGPs) for model set 2.

ment taking also the used heterogeneous *units* (e.g., ms, sec, min) into account.

Synopsis

As can be seen from the two model sets, our approach is capable of specifying, diagnosing, and handling a distinct set of consistency rules. In particular, the two model sets showed that the principle applicability of the approach is given for a set of distinct, heterogeneous engineering models. As the model sets consist of different modeling languages covering domain-specific modeling languages, general-purpose modeling languages, tool-specific modeling languages (such as MATLAB/Simulink), and task-specific modeling languages (such as QN for performance evaluation), we argue that a broad range of heterogeneous engineering models can be covered by our inconsistency management framework. Therefore, our approach is appropriately expressive to cover models and inconsistencies for automated production systems engineering (see Research Question 1).

With PaMoMo, we allow for two types of inconsistency specification: a graphical part, which allows for graphically specifying the inconsistency patterns as well as a textual part, which allows for extending the patterns with OCL-based first order logics. Obviously, the expressiveness of PaMoMo's graphical parts is limited and, therefore, less than the expressiveness of first order logic. For instance, PaMoMo does not allow for modeling the absence of cycles of given relations. Nevertheless, by means of OCL, we can also specify and compute the transitive closure for a relation (e.g., to identify whether cycles in relations are absent or not).

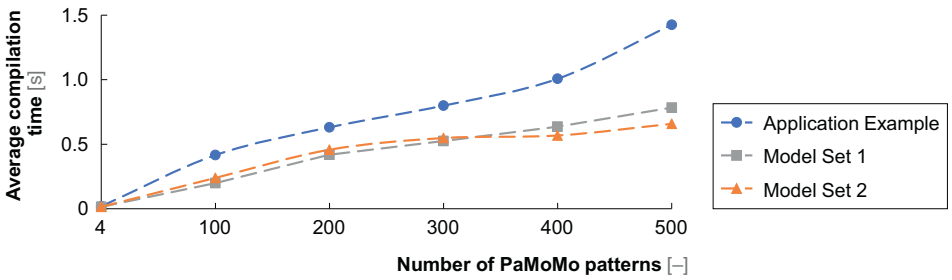
Table 1
Overview of comparison of PaMoMo patterns (in terms of Pattern Element Count (PEC)) against generated EVL code (in terms of Lines of Code (LoC)).

Case	PEC	EVL	Ratio PEC vs. LoC
Application example	35	800	4.4%
Model set 1	42	460	9.1%
Model set 2	37	640	5.8%
Overall	114	1900	Average 6.0%

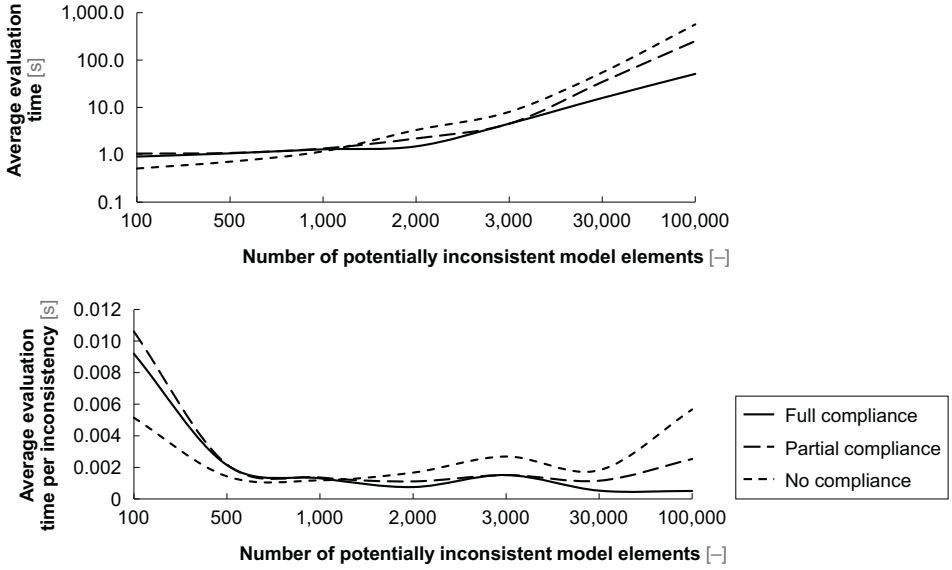
6.3.2. Efficiency: comparing pamomo patterns against generated EVL code

We now discuss the *efficiency* of our framework. In particular, by comparing the size of the resulting PaMoMo patterns with their generated counter-parts on the code level for both model sets, we provide indications for the efficiency of our approach. For the PaMoMo patterns, we use the Pattern Element Count (PEC) metric, which corresponds to the number of elements needed to describe the constraint. For the EVL code level, we resort to the classical Lines of Code (LoC) metric.

As can be seen in Table 1, approximately 40 pattern elements must be modelled for each of the investigated model sets (i.e., PEC



(a) Average compilation time depending on number of PaMoMo patterns



(b) Average evaluation time depending on number of potentially inconsistent model elements – both overall evaluation time (top) and evaluation time per inconsistency (bottom)

Fig. 16. Performance analysis for compilation and evaluation of PaMoMo patterns.

value of 35 for application example, 42 for model set 1, and 37 for model set 2). In contrast, on EVL code level, approximately 600 lines of code must be developed for each of the investigated model sets (i.e., LoC value of 800 for application example, 460 for model set 1, and 640 for model set 2). As a consequence, comparing the effort required to model the PaMoMo patterns (PEC value) with the effort to specify consistency constraints on an EVL level (LoC value), the comparably low effort to specify consistency constraints by means of PaMoMo is obvious.

Therefore, it can be concluded that our inconsistency management approach is more efficient compared to manual coding of the required consistency rules (see Research Question 2). Certainly, our comparison only provides an indication of the reduced effort, as the comparison is based on a comparison between (manually) specified PaMoMo patterns and (automatically) generated EVL code. However, we argue that this trend is sufficiently indicative for the improved efficiency of our approach compared to manually specifying consistency constraints on an EVL code level. This is also further investigated through our experiment introduced in Section 7.

6.3.3. Scalability: compilation times and execution times

This section provides a first indication of the *scalability* of our inconsistency management framework (see Research Question 3). In particular, we identify how the execution time (*i*) to generate EVL code from PaMoMo patterns and (*ii*) to diagnose inconsistencies and generate handling actions performs with an increasing number of model artefacts and consistency constraints.

In a first experiment (see Fig. 16a), we investigate the performance of the PaMoMo to EVL compiler by means of three sets of input models – one for the *application example*, one of *model set 1* and one for *model set 2*. In the experiments, the compiler's execution time is measured for different numbers of PaMoMo patterns (from 4 to 500 patterns), which are duplicated from the PaMoMo patterns introduced for our model sets. Therein, for each number of patterns, 10 compilation runs are performed and the average of all runs is used for comparison. As can be seen from the results in Fig. 16a, the compiler's execution time increases linearly with an increasing number of PaMoMo patterns. As the PaMoMo patterns for our application example are more complex than the ones defined in our model set 1 and 2, the highest compilation time can be identified for the application example. However, with a compilation time of approximately 1.5 s for 500 PaMoMo patterns in the application example, we argue that the compiler scales appropriately also for more complex models.

In a second experiment (see Fig. 16b), the performance of the constraint evaluation is measured in three scenarios – a *full compliance* scenario, in which all links are present and, hence, no inconsistencies arise, a *partial compliance* scenario, in which 50% of the links are present and, hence, 50% of the links are inconsistent, and a *no compliance* scenario, in which no links have been employed between the entities that should be linked. All three scenarios are applied for test models with approximately 100,000 internal elements, which are linked to a varying number of overlapping (and, hence, potentially inconsistent) elements. As can be seen from Fig. 16b, the evaluation time increases

linearly for fully compliant models and polynomial for the others with an increasing number of potentially inconsistent model elements.

Accordingly, we conclude that our inconsistency management approach is able to scale with an increasing number of modelled entities as well as specified inconsistencies (see Research Question 3). Certainly, our experiments only provide an indication of the framework's scalability, as the experiments are based on synthetic sets of input models. We, nevertheless, argue that this trend is sufficiently indicative for the scalability of our approach with an increasing number of models and inconsistencies. For model sets fulfilling most of the consistency rules, a fast evaluation can be guaranteed to work on-line. If no compliance is given, more time is needed to reach a first integration of the models which may require to run a batch process off-line.

6.4. Threats to validity

In this concluding section, we discuss the factors that may jeopardize the validity of our results.

Internal validity – Are there factors which might affect the results in the context of our model sets?

With our evaluation, we showed how the presented inconsistency management framework can be applied to two distinct model sets. These model sets comprise of typical models in the automated production systems domain that have been defined for a lab-scale demonstration case. However, the main question remains whether these models are the correct ones – namely whether these models are representative enough to consider them as realistic for industry applications in the automated production systems domain. Based on our evaluation, we assume that our approach is expected to be applicable and sufficiently scalable also for more complex applications. This fact must be investigated in future experiments. Furthermore, consistency rules were derived from different sources – both from guidelines and heuristics that are common to the domain of automated production systems and from the experience we gained in the field of inconsistency management. However, whether these consistency rules are representative enough for industrial applications must be verified in future work and in discussion with industry experts.

External validity – To what extent is it possible to generalize the findings for production system engineering in general?

With the proposed implementation, the specification and automatic diagnosis of inconsistencies as well as generation of potential handling actions is possible. The assumption for any model to be incorporated in our inconsistency management approach is that it must be MOF-based – which, however, is mostly the case for standard modeling languages and tools such as UML, SysML and MATLAB/Simulink. Nevertheless, this also introduces several shortcomings and risks: First, non-MOF-based models (e.g., engineering documents formulated in textual formats) are difficult to be incorporated as they need further preprocessing, e.g., by means of text mining and automatically creating graph-based models. In such cases, it has to be estimated whether the additional effort is appropriate compared to the benefits of automated inconsistency management. Second, the effort to define the intensional links between the different models to be incorporated during inconsistency management can be huge, especially if a multitude of different modeling formalisms and/or tools is being used. Specifically, if n models are involved during inconsistency management, $n \cdot (n-1)/2$ link models must be defined. Thus, the need arises to provide abstraction mechanisms that, e.g., mediate the models into a common semantic level and, thereby, allow to reduce the number of necessary link models to n. Third and finally, our inconsistency management approach stands and falls with the knowledge encoded within the consistency rules. Although we assume that a graphical specification of the consistency rules can help in capturing this knowledge, it can be assumed that the effort in training domain experts in this formalism can be large. However, we argue that PaMoMo, as a graphical language to declaratively specify the dependencies between model elements, serves as the means to reduce the effort.

Summary

Summarizing the discussions regarding threats to validity of our inconsistency management framework, we hypothesise that our framework is a valid basis for future work in the field of inconsistency management, especially for practical applications in the domain of automated production systems. Further experiments are planned to validate its applicability, also for more complex industrial system setups.

7. Empirical experiment

In order to verify the results we derived from our case study evaluation, a small empirical experiment has been conducted following the guidelines presented in Ko et al. (2015). Therein, participants were asked to create and maintain different consistency rules by means of two different approaches – (*i*) by specifying consistency rules textually by means of EVL and (*ii*) by specifying consistency rules graphically by means of the PaMoMo-based approach presented in this article. It has to be noted that with a small amount of 8 participants from the Technical University of Vienna we aim at a first qualitative study of how this approach is perceived by potential end users.

Within this section we introduce this experiment and illustrate the findings we obtained. In particular, we first introduce the experiment's objectives in Section 7.1. Second, we show our experiment set-up – namely the participants, materials and tasks (see Section 7.2). The results of our experiment are discussed in Section 7.3. Finally, we discuss the threats to validity of our results in Section 7.4.

All the documents we used throughout our experiment can be obtained online.[11]

7.1. Objectives of the experiment

The experiment was set up as a controlled experiment where participants have used the two approaches – namely EVL and PaMoMo – to specify and maintain distinct consistency rules. The objectives of the experiment were:

- Analyze: the presented inconsistency management approach using PaMoMo as well as EVL,
- With the goal of: evaluating our proposed inconsistency management approach,
- In terms of: effectiveness and efficiency for specifying and maintaining consistency rules,
- From the point of view: of end users, and
- Related to (*i*) specifying consistency rules from scratch as well as (*ii*) maintaining already specified consistency rules.

Therein, we follow the initial results of our case study evaluation that indicated increased effectiveness and efficiency while using PaMoMo instead of EVL. Using the experiment, we would like to find an indication whether our initial hypothesis (that PaMoMo proves to be more effective and efficient for specifying and maintaining inconsistencies compared to EVL) holds true.

[11] https://mediatum.ub.tum.de/1356820, retrieved January 20, 2019.

7.2. Experiment set-up

Participants

A group of 8 people has participated in this controlled experiment. These people were all students and/or faculty members at the Technical University of Vienna. We ensured a mix of 5 female and 3 male participants as well as a broad variety in their study background including business informatics, informatics, architecture as well as media technology and design. Certainly, with a small amount of 8 participants we focus on a first indication of how this approach is perceived by potential end users.

By means of a self-assessment survey, the participants provided their (subjective) view regarding their previous knowledge in related fields – hence, we ensured that similar basic knowledge in the fields of modeling and meta-modeling was present for all participants. Although being experienced users in these fields, they had none or only basic knowledge in the field of inconsistency management.

Procedure

For our experiment, we chose a three-step procedure: In *step 1*, we introduced the experiment objectives and collected the participants' background data such as gender, field of study and current level knowledge (approximately 10 min). In *step 2*, we introduced both approaches – namely the inconsistency management approach using EVL and the one using PaMoMo – at the hand of a simple application example (approximately 20 min). In the actual experiment *step 3*, we introduced the participants into the experimental models (approximately 10 min) and asked them to perform 4 tasks – 2 tasks that involve independent creation of consistency rules and 2 tasks that involve maintenance of consistency rules. We set a time frame for working on these tasks of 20 min to see what can be achieved within such a tight time frame with the two approaches. To perform the experiment, we split our participants into two groups: one group of 4 people working with EVL and another group of 4 people working with PaMoMo. Through the splitting of the groups, we aimed at ensuring that no learning effects are introduced. The assignment of people to groups was random. We closed step 3 with a survey that follows the suggestion of a System Usability Scale (SUS) by Brooke (1996). Using this SUS, we aim at identifying, how the participants perceived working with the respective approach.

All material – including surveys, tasks as well as introductory documents – can be obtained online.[7]

Tasks

As indicated above, the participants were asked to perform 4 tasks within approximately 20 min, of which 2 involved the creation of consistency rules (i.e., creating consistency rules from scratch) and 2 involved the maintenance of consistency rules (i.e., finding errors within already specified consistency rules). All tasks were related to the model set 1 introduced in our case study evaluation (see Section 6) and included the types of consistency rules that have been introduced for this model set.

To ensure that all participants (both in the EVL and PaMoMo group) have the same basic information, the tasks were designed in a way that participants needed to perform the identical actions on EVL and PaMoMo side – i.e., they needed to specify the identical consistency rules on both sides. For the tasks that involved the creation of consistency rules, this means that both groups received the same background information for specifying the rules. For instance, Fig. 17a illustrates the task for creating a consistency rule to verify the RefinesLinks between modules and blocks. As can be seen from the figure, identical steps needed to be performed in order to specify this rule. For the tasks that involved the maintenance of consistency rules, both EVL and PaMoMo groups needed to identify the same errors (e.g., wrong variable navigations, wrong classifiers, etc.). For instance, Fig. 17b illustrates the task for maintaining a consistency rule to verify the hierarchy equivalence between modules and blocks. The figure shows that on both sides, the identical errors needed to be found by the participants.

Survey

A survey aimed at identifying the participants' view on the usability of the approaches.

This post-assessment was structured along the 10 statements of the System Usability Scale (SUS) introduced by Brooke (1996). Therein, the participants were asked to assess whether they agree or disagree to the respective statements after having used the respective approach.

7.3. Results

Task results

We analysed the participants' results regarding the experiment tasks along two dimensions: (*i*) the *completeness* of the results has been measured by means of the number of tasks that the participants completed within the given time frame of approximately 20 min, and (*ii*) the *correctness* of the participants' results has been measured by a simple rating scheme based on the blueprint solution[12].

The results are illustrated in Fig. 18. As can be seen from Fig. 18a, the participants that used PaMoMo were able to complete more tasks within the given 20 min than the participants that used EVL. Whereas all participants using PaMoMo were able to solve at least tasks 1–3, none of the participants using EVL was able to complete task 3 within the time frame. Consequently, participants of the EVL group ran out of time while solving the tasks. One conclusion could be that using PaMoMo, the participants within our experiment can create and maintain consistency rules more efficiently than using EVL. A second conclusion could be that EVL participants struggled with comprehension problems. In both cases, indication is provided that PaMoMo is understood easier by participants compared to EVL. In addition, as can be seen from Fig. 18b, PaMoMo lead to more correct results in the experiment tasks. Whereas the results for the creation tasks 1 and 2 were only slightly better, the maintenance of consistency rules in tasks 3 and 4 were easier according to the experiment results. Please note that, although none of the participants was able to finish tasks 3 and 4, there were some participants that have worked on task 3 – hence, a correctness result of $>0\%$[13] was achieved.

Survey results

Analogously, we used the survey to analyse the participants' perception of the usability of both approaches.

Hence, a post-assessment served as survey to assess the usability after using the approach along the statements introduced for the SUS by Brooke (1996) (see results in Fig. 19). As can be seen from the results, the participants see PaMoMo especially helpful

[12] For the tasks that demanded *creating* a consistency rule, in total 5 points could be achieved: all links being checked, all meta classes being checked, all properties being checked, correct syntax, correct global conditions (if available). For the tasks that demanded *maintaining* a consistency rule, in total 4 points could be achieved: one point for each error identified (in total 4 errors in both tasks). The completeness score for the analysis of the results was determined based on the participants' share of points.

[13] Average correctness value calculated based on scores for partly or fully solved solutions.

Task 1 description: "Refines-links between Modules and Blocks: Creation of an inconsistency pattern"

Please formulate the consistency rule for the following description:

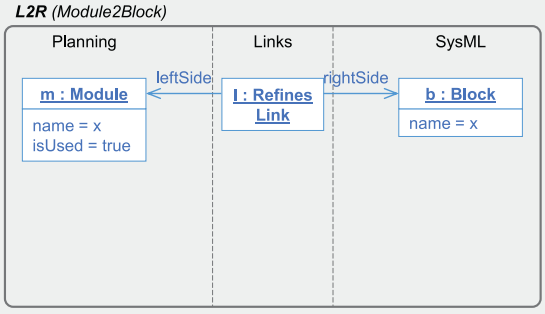
Each Module in the Planning Model that is selected to be used for the engineering solution (i.e., value of property "isUsed" is equal to "true") must be linked to an equivalently named Block in the SysML model (i.e., there must be a Refines-link that points to a Module in the Planning Model and a Block in the SysML model with equivalent "name" properties).

XX – Prefilled elements, XX – Blueprint solution

EVL group

```
context planning!Module {
  constraint ModuleNeedsBlock {
    check {
      return RefinesLink.allInstances()->
        exists( l | l.rightSide = self and
          l.leftSide.isTypeOf( sysml!Block ) and
            l.leftSide.name = self.name );
    }    /* ... */
  }
}
```

PaMoMo group



(a) Overview of task 1: Creating a consistency rule for the refines link between modules and blocks

Task 4 description: "Consistency of module hierarchies between Modules and Blocks: Maintenance of an inconsistency pattern"

A consistency rule was formulated for the following description:

Each top-level Simulink Display must satisfy a Property in the Planning Model (i.e., for each top-level Display, there must also exist a SatisfiesLink to a Property in the Planning Model). In this case, the values of the properties must be consistent (i.e., the value of the Display must be within the minimum and maximum range of the Property and identical to the default value of the Property). For the sake of simplicity, we assume that only one link between the Display and the Property exists.

The following specification contains several errors. Please correct them.

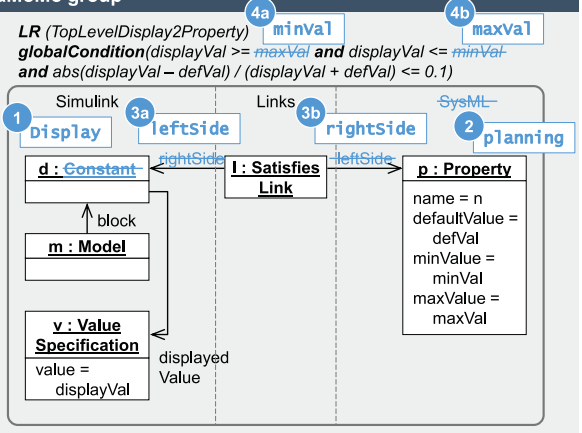
XX – Prefilled elements, XX – Blueprint solution

EVL group

```
context simulink!~~Constant~~ {
  constraint TopLevelDisplayNeedsProperty {
    check {
      var link : link!SatisfiesLink.allInstances() ->
        select( l | l.~~rightSide~~ = self and
          l.~~leftSide~~.isTypeOf(~~sysml~~!Property)).first();
      var minVal = link.~~leftSide~~.minValue;
      var maxVal = link.~~leftSide~~.maxValue;
      var defVal = link.~~leftSide~~.defaultValue;
      var displayVal = self.value;
      return displayVal >= ~~maxVal~~ and displayVal <= ~~minVal~~ and
        ((displayVal-defVal).abs()/(displayVal+defVal)) <= 0.1;
    }    /* ... */
  }
}
```

1 Display · 2 planning · 3a leftSide · 3b rightSide · 4a minVal · 4b maxVal

PaMoMo group



(b) Overview of task 4: Maintaining a consistency rule for the hierarchy equivalence between modules and blocks

Fig. 17. Overview of two sample tasks with the user experiments.

regarding its simplicity and would, hence, use this approach frequently. However, what remains open from the participants' perspective is that assistance is needed to use our approach (see ④ in Fig. 19) and that the aspects and functionalities of our approach should be better integrated (see ⑤ in Fig. 19). With adequate tool support, these aspects can be overcome in the future. By applying Brooke's SUS calculation scheme, we obtain an average SUS of 69 for PaMoMo and of 39 for EVL. However, "[a]s with any metric, the SUS score should not be used in isolation to make absolute judgements [...]" (Bangor et al., 2008). Hence, also showing obviously better results than the EVL approach, we see that there are still improvement potentials to our PaMoMo-based approach: according to Bangor et al. (2008), the PaMoMo approach is still only in the 2nd quartile.

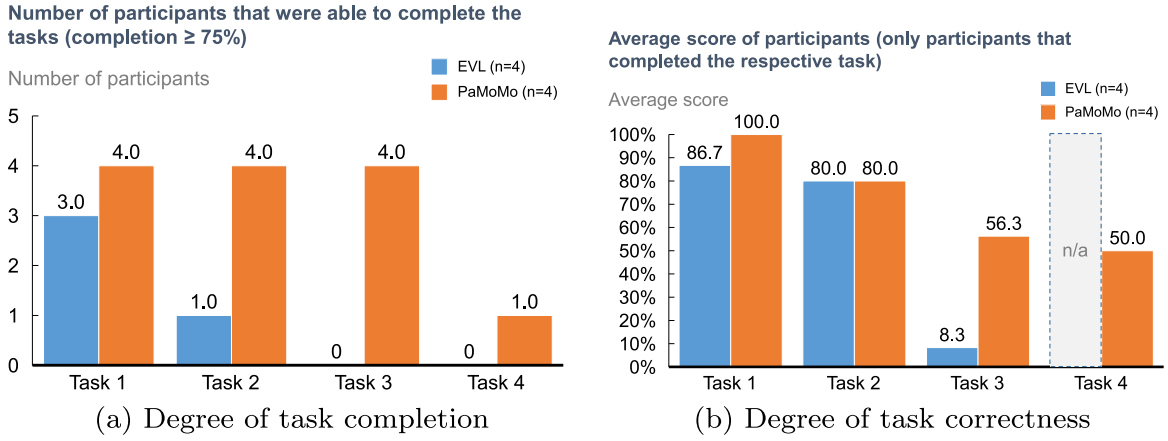


Fig. 18. Overview of the experiment task results.

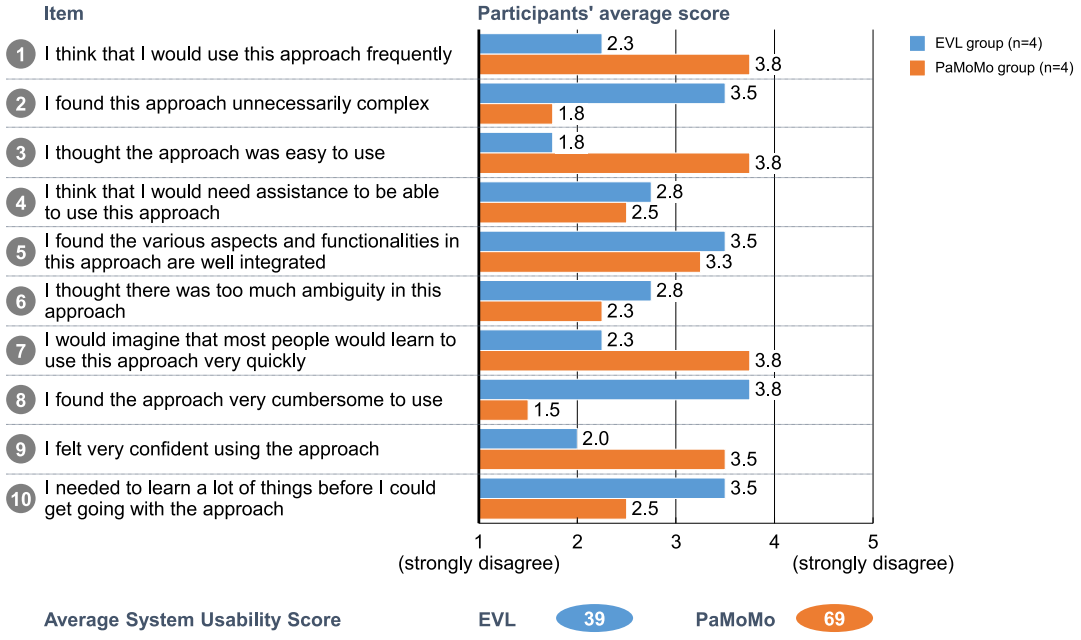


Fig. 19. Overview of the participants' assessment of the approaches' usability after the experiment.

7.4. Threats to validity

In this concluding section, we discuss the factors that may jeopardize the validity of our results.

Internal validity – Are there factors which might affect the results in the context of our experiment?

For performing the experiment, we used a lab-size model set introduced in Section 6 and asked the participants to model the respective consistency rules for this lab-size model set. With our experiment, we showed that PaMoMo seems to be more effective and efficient for modeling and maintaining these consistency rules. This fact must be investigated in future experiments – especially to validate whether this still holds true for more complex model sets and consistency rules.

External validity – To what extent is it possible to generalize the findings of our experiment for production system engineering in general?

The experiment involved a small set of participants – namely 8 students and/or faculty members at the Technical University of Vienna. Consequently, we do not aim at results with a high significance, but rather at a first indication of whether our approach is applicable for specifying consistency rules in an effective and efficient means. Hence, next steps will require to validate the results in a broader context, involving more participants from a variety of fields of study (also including, e.g., the application domain of automated production systems) as well as from industry.

Summary

Summarising our findings, with the experiment we showed that our PaMoMo-based framework seems to be more effective and efficient for modeling and maintaining consistency rules compared to approaches based on textual languages such as EVL. Certainly, further experiments are required to validate the applicability and usability of our approach, especially for more complex systems in industrial settings.

8. Related work

This section is devoted to discussing the research related to this article. We will first focus on model-based (systems) engineering approaches (see Section 8.1), which form the basis for

managing inconsistencies. Second, inconsistency management approaches (see Section 8.2) will be discussed, which provide the technological means to specify, diagnose and handle inconsistencies within and in between engineering models. Finally, our findings are summarized in Section 8.3.

8.1. Model based (systems) engineering approaches

Model-based systems engineering (MBSE) constitutes a shift from a pure document-centric view towards the use of integrated models. However, different disciplines make use of disparate modeling formalisms, abstraction levels and tools (Broy et al., 2010; Gausemeier et al., 2009). Consequently, MBSE approaches need to incorporate these different aspects in an adequate manner. To allow for a holistic view on these different disciplines, two main approaches can be identified in the literature: (*i*) the use of integrated models, in which a system model is used to consistently manage the different, discipline-specific aspects of the system and (*ii*) the application of (bidirectional or unidirectional) model mappings between the different discipline-specific models. These two approaches as well as their benefits and limitations are discussed in detail in the following.

8.1.1. Integrated system models

Following the MBSE paradigm, an integrated model provides a holistic view on the system. MBSE requires, next to a suitable modeling language, also an appropriate design methodology (Barbieri et al., 2014), which guides the stakeholders and assures an efficient engineering process. Arising from software engineering, some MBSE approaches (e.g., Schäfer and Wehrheim, 2010) are based on UML. Therein, UML has proven its applicability for software engineering in the automated production systems domain (Vogel-Heuser, 2015). The use of SysML for the integrated development of complex manufacturing systems has shown that a model hierarchy with different levels of detail (Bassi et al., 2011) or different views within the integrated model (Thramboulidis, 2013) are necessary to capture all required information appropriately. Another approach for the integration of discipline-specific information is the creation of a knowledge base, which integrates all data from the different specific models and supplies transformation rules (Moser and Biffl, 2012). By that, engineers can use the tool they are familiar with and store the model in the knowledge base. However, as no integrated visual model is present, overlaps between different models can easily be overseen, resulting in potential inconsistencies which, of course, should be managed.

8.1.2. Model mapping and linking support

One major requirement for inter-model consistency approaches is the non-intrusive linkage of models. In order to keep the models as they are used in their particular domains, one frequent approach is to employ different kinds of link models (Bézivin et al., 2006a; Aizenbud-Reshef et al., 2006). Several dedicated approaches have been proposed for model weaving (Jossic et al., 2007), model merging (Kolovos et al., 2006), model traceability (Maro and Steghöfer, 2016), and model integration (Schürr and Klar, 2008). Please note that for model traceability, there is a recent approach (Maro and Steghöfer, 2016) for creating trace links between heterogeneous artefacts, defining specific trace link types as well as tracking and visualizing trace links inside Eclipse. However, in the current state of Maro and Steghöfer (2016), no dedicated pattern language is provided as we have defined in the context of this paper with PaMoMo to diagnose inconsistencies. For model integration, several dedicated approaches have been proposed in the past how to deal with inconsistencies, e.g., (Finkelstein et al., 1994; Eramo et al., 2008; Bergmann et al., 2012). In the next section, we discuss these inconsistency management approaches in detail.

8.2. Inconsistency management approaches

Although core MBSE concepts and techniques such as integrated system models and linking support provide an appropriate basis for inconsistency management, appropriate concepts and techniques for managing inconsistencies need to be put in place. The term *inconsistency management* involves three essential components (Nuseibeh et al., 2000):

- the means to *specify* the consistency rules (that is: the constraints that must be met by the models) to be managed,
- the means to *diagnose* inconsistencies through continuously locating, identifying and classifying them, and
- the means to *handle* inconsistencies by proper actions such as ignoring, tolerating or resolving the inconsistencies.

In previous work (Feldmann et al., 2015b), we compared existing approaches that aim at managing inconsistency in sets of heterogeneous models. We discovered that a multitude of such approaches exists and classified them into (*i*) approaches that make use of *logical reasoning and theorem proving*, in which formal models are used to formally conclude whether the models are consistent or not, (*ii*) *rule- and pattern-based approaches*, in which (negative or positive) constraints are used to describe the conditions for the presence of an (in-)consistency, and (*iii*) *synchronization-based approaches*, in which bi-directional or uni-directional model synchronizations are used to propagate changes from one model to another. These three different approaches are discussed in the following.

8.2.1. Logical reasoning and theorem proving

Within approaches that make use of logical reasoning and theorem proving, a well-defined formal system is used, in which inconsistencies can be identified.

For instance, Finkelstein et al. (1994) make use of first-order logic to diagnose inconsistencies within multi-view software models such as class diagrams, sequence diagrams, etc. By means of domain-specific rules, which are specified in temporal logic, inconsistencies can be resolved. Within the work of Schätz et al. (2003), a first order propositional logic similar to OCL is used to formulate consistency conditions and to describe the underlying formal model. UML models, in particular class diagrams, state charts and sequence diagrams, are in focus of Van Der Straeten et al. (2003). Using a rule-based approach (Mens et al., 2005), inconsistencies can be resolved (semi-)automatically.

There are also some existing approaches using constraint solving for model synchronization such as the Janus Transformation Language (JTL) (Cicchetti et al., 2011), also with compressed state spaces (Eramo et al., 2015), and the CARE approach (Schoenboeck et al., 2014). JTL (Cicchetti et al., 2011) uses answer set programming (ASP) to find a synchronized result. In previous work, we have presented CARE (Schoenboeck et al., 2014), an approach using a constraint solver to re-synchronize models with their evolving metamodels. However, CARE is a specific approach for the metamodel/model co-evolution problem. Furthermore, transformation models using UML/OCL have been already introduced back in 2006 (Bézivin et al., 2006b) which may be also exploited for automatically synchronizing models by employing model finder tools in addition with costs for synchronization solutions (Bill et al., 2016).

Dávid et al. (2016, 2017) focus their research on reasoning about inconsistency tolerance by “quantifying the impact of single inconsistencies on the overall system design” (Dávid et al., 2016). In particular, they distinguish between so-called *parameter* tolerance (which allows for slight deviations from parameter values) as well as *temporal* tolerance (which allows inconsistencies for a certain amount of time). These two types of tolerance serve as the main

motivation for our tolerating handling actions, which we extended with the notion of ignoring handling actions (to explicitly stating that an inconsistency is no longer of interest to the user).

Whereas the benefits in logical reasoning and theorem proving approaches lie in the formality (i.e., it can be formally concluded whether the models are consistent or not), the effort and required expert knowledge to formulate the necessary formal models is tremendous. Hence, especially for the multitude of disciplines in the automated production systems domain, it can be concluded that such approaches are not cost-effective in their application.

8.2.2. Rule- and pattern-based approaches

Similarly to approaches that make use of logical reasoning and theorem proving, rule- and pattern-based inconsistency management aims at applying a rule or pattern base that describes either the sufficient conditions that a model must satisfy for it to be considered consistent (Hehenberger et al., 2010) – that is, rules are used as *positive constraints* – or as *negative constraints*, which represent the sufficient conditions that indicate an inconsistency (Herzig et al., 2014). Accordingly, *patterns* can indicate whether an inconsistency exists or not. However, what makes the approaches different from logical reasoning and theorem proving is that, instead of targeting a pre-defined, complete and consistent formal system, the knowledge base in rule- or pattern-based approaches is always incomplete (Nuseibeh et al., 2000): Rules or patterns can be added and/ or removed without the need to rethink (and – in a worst case – adapt) the entire knowledge base.

Positive constraints

Egyed (2011) aim at diagnosing and tracking inconsistencies within software engineering models. Mechatronic system models are focused on in Hehenberger et al. (2010). By means of a unifying and domain-spanning mechatronic ontology, model elements are tagged to implicitly specify the links between different models in a common, well-defined syntax and semantics. In addition, Muskens et al. (2005) introduce a rule-based approach for diagnosing inconsistencies between UML diagrams by means of relation partition algebra. In particular, they aim at diagnosing violations of *lower bounds* (i.e., *obligations*) and *upper bounds* (i.e., *constraints*). Whereas such *obligations* require that a subordinate UML diagram must contain at least the information modelled in a prevailing diagram, *constraints* require a subordinate diagram to contain only the information available in the prevailing diagram. With their approach they cover both intra- and inter-development phase consistency checking – hence, a multitude of applications in the UML-based development of software systems can be found. However, none of the aforementioned approaches investigates the handling of diagnosed inconsistencies.

Negative constraints

One representative for a rule-based inconsistency management approach that makes use of negative constraints is the work of Mens et al. (2006). By means of critical pair analysis, Mens et al. (2006) are able to identify both parallel and sequential dependencies between resolution rules, i.e., whether the rules are mutually exclusive (parallel dependency) and whether the rules have causal dependencies (sequential dependency). The results of the analysis provide a basis to present appropriate resolution rules to the users. Nonetheless, these approaches are mainly focused on software models – their application to heterogeneous engineering models has not yet been investigated.

Pattern-based approaches

Within Hegedüs et al. (2011), so-called *quick fixes* are suggested as the means to diagnose and resolve inconsistencies in domain-specific languages. Quick fixes originate from the domain of software engineering, and are "[...] a very popular feature of integrated development environments [...], which aid programmers in quickly repairing problematic source code segments" (Hegedüs et al., 2011). Especially Semantic Web Technologies are being applied for the purpose of inconsistency management. For instance, Herzig et al. (2014) propose a framework for the diagnosis of inconsistencies, in which graph patterns are used to define inconsistency diagnosis rules. Further applications of such Semantic Web Technologies can be found in Feldmann et al. (2014, 2016a) for the purpose of analysing the compatibility of different mechatronic system modules, in Abele et al. (2013) in which the Web Ontology Language (OWL) is used together with SPARQL to analyse pre-defined well-formedness constraints as well as in Kovalenko et al. (2014) and Biffl et al. (2014) for an ontology-based cross-disciplinary defect detection. Nonetheless, none of these approaches aim at interactively handling the diagnosed inconsistencies.

8.2.3. Synchronization-based approaches

Contrary to the approaches described above, model synchronizations aim at unidirectional or bidirectional transformations between the models involved in the engineering process. Consequently, in such approaches, transformation rules are formulated that capture, how entities in one model are related to entities in another model. An inconsistency can then be regarded as a state of conflict that is not reproducible by executing the transformation rules. Hence, synchronization-based approaches are similar to rule-based approaches as a set of rules forms the basis for the purpose of managing inconsistencies between heterogeneous models.

For instance, triple graph grammars (Schürr, 1994; Kindler and Wagner, 2007) are often employed for model synchronization scenarios such as the ones reported in Anjorin et al. (2014), Hermann et al. (2012) and Giese and Wagner (2009). In a similar fashion, QVT Relational (Stevens, 2009) allows synchronization, also in conjunction with unidirectional transformation languages such as ATL (Macedo and Cunha, 2014). Furthermore, there are other, mostly rule-based, approaches available (cf. Hidaka et al., 2016 for a survey). However, a variety of necessary correspondence rules is expected for the multitude of engineering models in the automated production systems domain – consequently, the effort to create and maintain these rules increases with the number of model types that need to be considered.

8.3. Synopsis

To sum up, all these mentioned approaches lack support for managing inconsistencies as they are either mainly focusing on user-defined links without automatic support (cf. Section 8.1) or they aim for a fully automatic generation and maintenance of links without any interaction possibilities for the engineers (cf. Section 8.2). In this paper, we aim to provide both: an extensible, semi-automatic support that allows for interaction possibilities in order to manage inconsistencies in a flexible way.

9. Conclusions and future work

This article introduced a comprehensive approach for managing inter-model inconsistencies in automated production systems engineering. We extended our previous work presented in Feldmann et al. (2016b) towards an extensible inconsistency management approach that builds upon four pillars:

- a link modeling language to define and elaborate links between heterogeneous engineering models,
- an extended graphical, pattern-based modeling language to define consistency rules,
- a framework with according prototypical tool support to automatically generate executable consistency rules based on the graphically modelled patterns, as well as

- two distinct industry-inspired model sets to indicate first findings regarding the specification capabilities and effort as well as scalability of our framework in the automated production systems engineering domain.

As discussed in Section 8, inconsistency management frameworks can be broadly divided into logical reasoning and theorem proving, rule- and pattern-based as well as model synchronization approaches. The proposed inconsistency management framework is classified as rule-based approach, which makes use of positive constraints that describe the necessary and sufficiently known conditions for model consistency. Hence, we argue that our framework allows for flexibly extending and reducing the set of consistency rules. As our approach relies solely on existing standards (i.e., MOF-based metamodels) and according implementations (i.e., the Eclipse Modeling Framework), our approach can be adapted to a broad variety of models in the automated production systems domain.

The initial findings from applying our inconsistency management framework to two industry-inspired model sets indicate that both the specification efforts as well as the scalability are appropriate for model-based engineering of automated production systems. In future work, we extend the scope of our evaluation towards industry-scale models to optimize and adapt our approach towards industry-sized modeling scenarios in the automated production systems domain. To make our approach applicable also to large-scale model sets, several practical considerations must be made. For one, our current approach assumes that models are loaded in total into our inconsistency management framework - which could face obstructions in case of large-scale industry models. This could, e.g., be overcome through directly communicating with the model source (e.g., by means of a data base driver). Moreover, our approach assumes that models are MOF compatible and modeled in open source tools such as the Eclipse Modeling Framework. Nevertheless, industrial settings mostly use closed source tools such as MagicDraw, IBM Rhapsody, etc. Therefore, a seamless integration between these closed source tools and our framework must be ensured. As our prototype is based on the Eclipse Epsilon framework, one starting point for ensuring this seamless integration in a performant way could be the model driver proposed by Zolotas et al. (2017).

In addition, we will analyse the need to further extend PaMoMo's expressiveness. Whereas we focus on the use of triple patterns (i.e., two models are linked by an intermediate model), n-ary patterns (i.e., linking multiple models with multiple intermediate models) are currently not support natively in PaMoMo and its compiler. One starting point to include these n-ary patterns could be the research work of Stevens (2017), to specify such n-ary relations within models.

The initial evaluation of our approach within a small user experiment only provides a preliminary discussion of the usability of our approach. Therefore, future work comprises the extended evaluation of our approach with expert users – both from academia and industry - to investigate, how potential users interact with out inconsistency management framework. This will especially be part of the Collaborative Research Center 768 – 'Managing Cycles in Innovation Processes'.

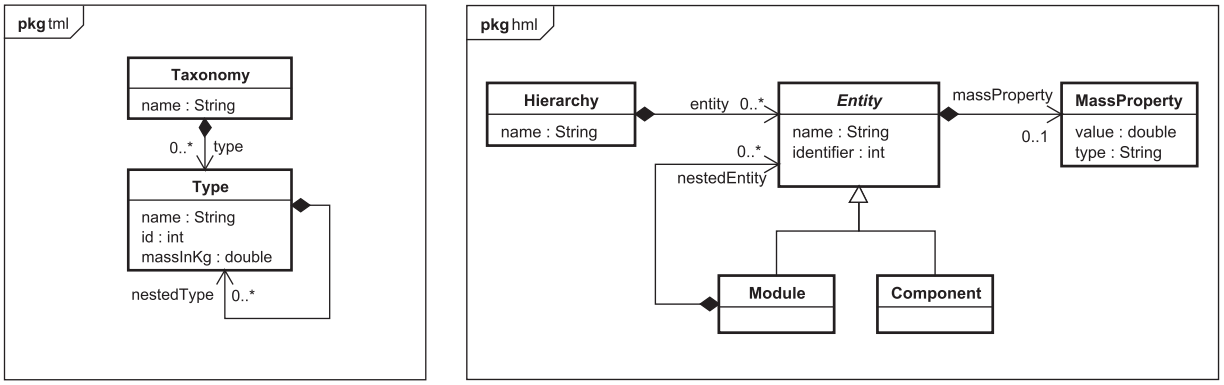
Moreover, the current compiler sequentially checks the modeled inconsistencies and creates respective diagnosis results. With explicitly modelling dependencies between inconsistency patterns, we ensure that basic patterns are statically checked before checking the more specific patterns to avoid, e.g., deadlocks in between the modelled inconsistency patterns. Nevertheless, the main drawback of this approach is that if inconsistency resolution actions are performed, these actions may cause new inconsistencies. Future work will, hence, involve the analysis of the potentials of incremental inconsistency detection for our inconsistency management framework. One starting point for this incremental inconsistency detection could be the approach proposed by Mens et al. (2006). By means of their clustering and ordering approach, respective dependencies between diagnosed inconsistencies and their handling actions could be analysed before applied to the models.

Finally, further directions for research are (*i*) integrating engineering documents such as CAD drawings and circuit diagrams into our inconsistency management approach, (*ii*) supporting "fuzzy" inconsistencies that are not focused on exact matches, but similarities, as well as (*iii*) integrating further inconsistency management aspects such as the concrete syntax of modeling languages into our inconsistency management approach.

Acknowledgments

We thank the German Research Foundation (DFG) for funding this work as part of the collaborative research centre 'SFB 768 – Managing cycles in innovation processes'. Moreover, this work has been partially funded by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and by the FWF under grant numbers P28519-N31 and P30525-N31. Finally, we thank the Technical University of Munich for funding parts of this works within the August-Wilhelm Scheer Visiting Professorship.

Appendix A. Metamodels of application example



(a) TML metamodel (b) HML metamodel

Fig. A.1. Metamodels of the application example.

Appendix B. EVL implementation

```
context tml!Type{
  constraint Type2Entity{
    check {
      var t = self;
      var X1 = self.name;
      var Y1 = self.id;
      var Z = self.massInKg;

      var l = allLinks()
        -> select(l|l.isKindOf(EquivalentToLink))
        -> selectOne(l| equals(l.leftSide, self) and
        l.rightSide -> exists(r| r.isKindOf(hml!Entity)));
      if(l == null){return false;}

      var e = l.rightSide;
      if(e  == null){return false;}

      var X2 = e.name;
      var Y2 = e.identifier;

      var m = e.massProperty;
      if(m  == null){return false;}

      var A = m.value;
      var B = m.unit;

      return X1 == X2 and Y1 == Y2 and Z.toBaseUnit("kg") == A.toBaseUnit(B);
    }
    message  {
      /* ... */
      var prev_inc = incMod.previousRun.getIncDiagnosisResult(self);
      if(prev_inc <> null){
        // inconsistency has been already handled before
        msg = 'The inconsistency concerning Type2Entity and ' + self +
          ' is currently: ' + prev_inc.status.name;
        inc = vr.createIncDiagnosisResult(msg,
          INCONSISTENCY!Severity#Information, self, 'context');
        inc.status = prev_inc.status;
      }else{
        // new inconsistency found
        msg = 'The constraint Type2Entity is not fulfilled for ' + self;
        inc = vr.createIncDiagnosisResult(msg,
          INCONSISTENCY!Severity#Error, self, 'context');
        inc.status = INCONSISTENCY!Handling#unhandled;
      }
      /* ... */
    }

    fix{
      guard : incMod.previousRun.getIncDiagnosisResult(self).status <>
                INCONSISTENCY!Handling#tolerated
      title : "Tolerate inconsistency"
      do{
        trace.get(self + 'Type2Entity').status =
          INCONSISTENCY!Handling#tolerated;
      }
    }
    /* ... */
  }
```

Listing B.1. Exemplary EVL implementation for inconsistency management.

Appendix C. Metamodels of model set 1

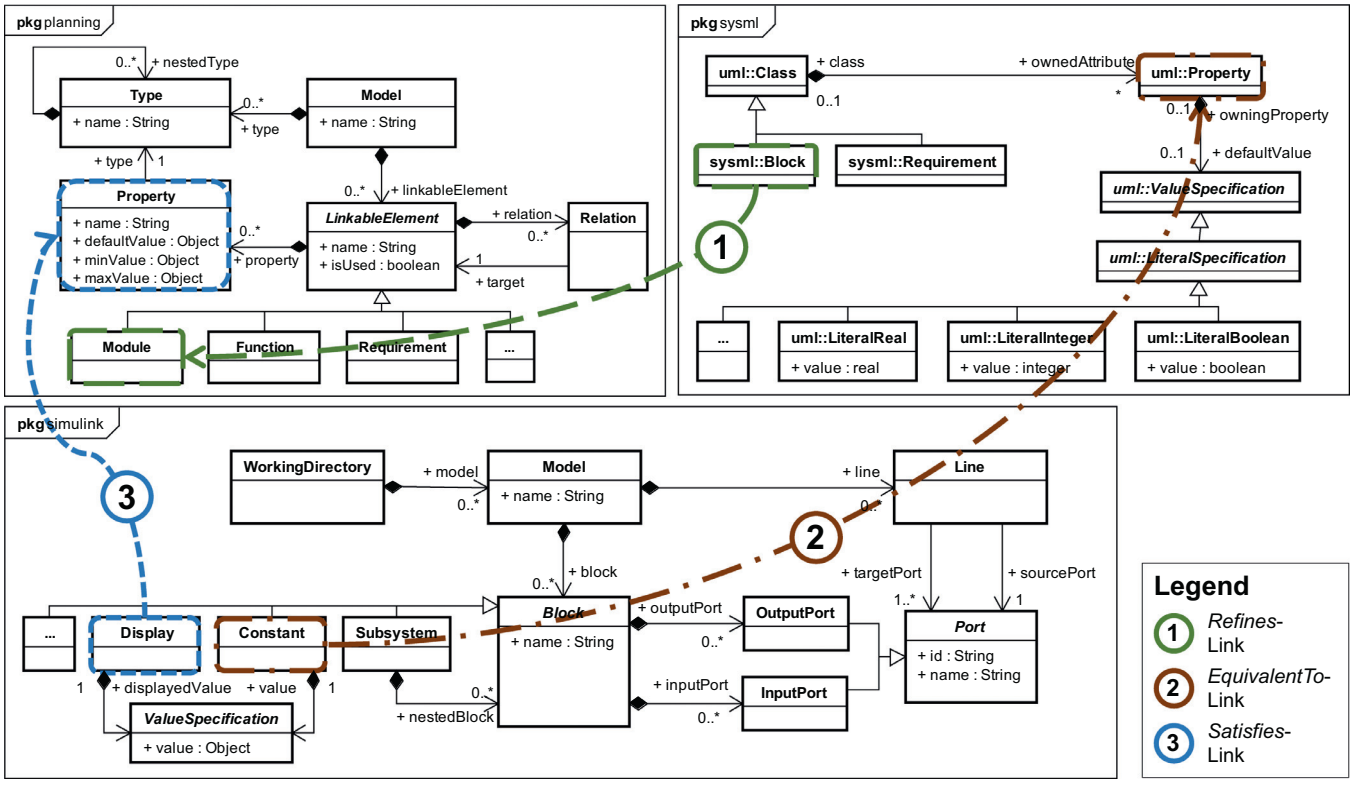


Fig. C.1. Overview of the metamodels in model set 1 (taken from Feldmann et al. (2016b)).

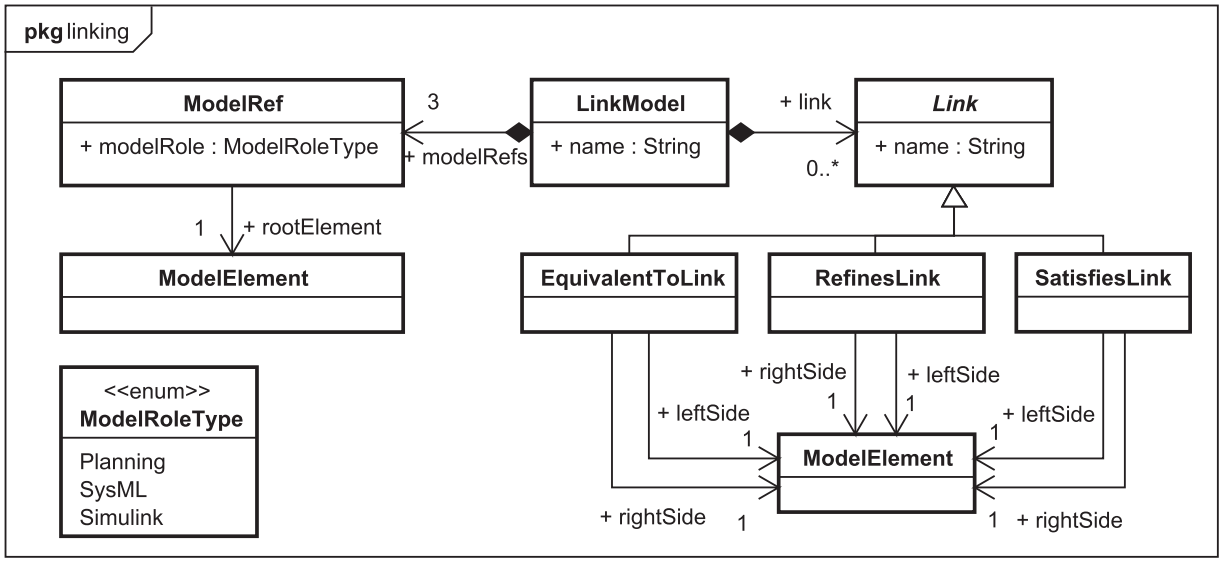


Fig. C.2. Linking language for model set 1 as MOF-based metamodel.

Appendix D. Metamodels of model set 2

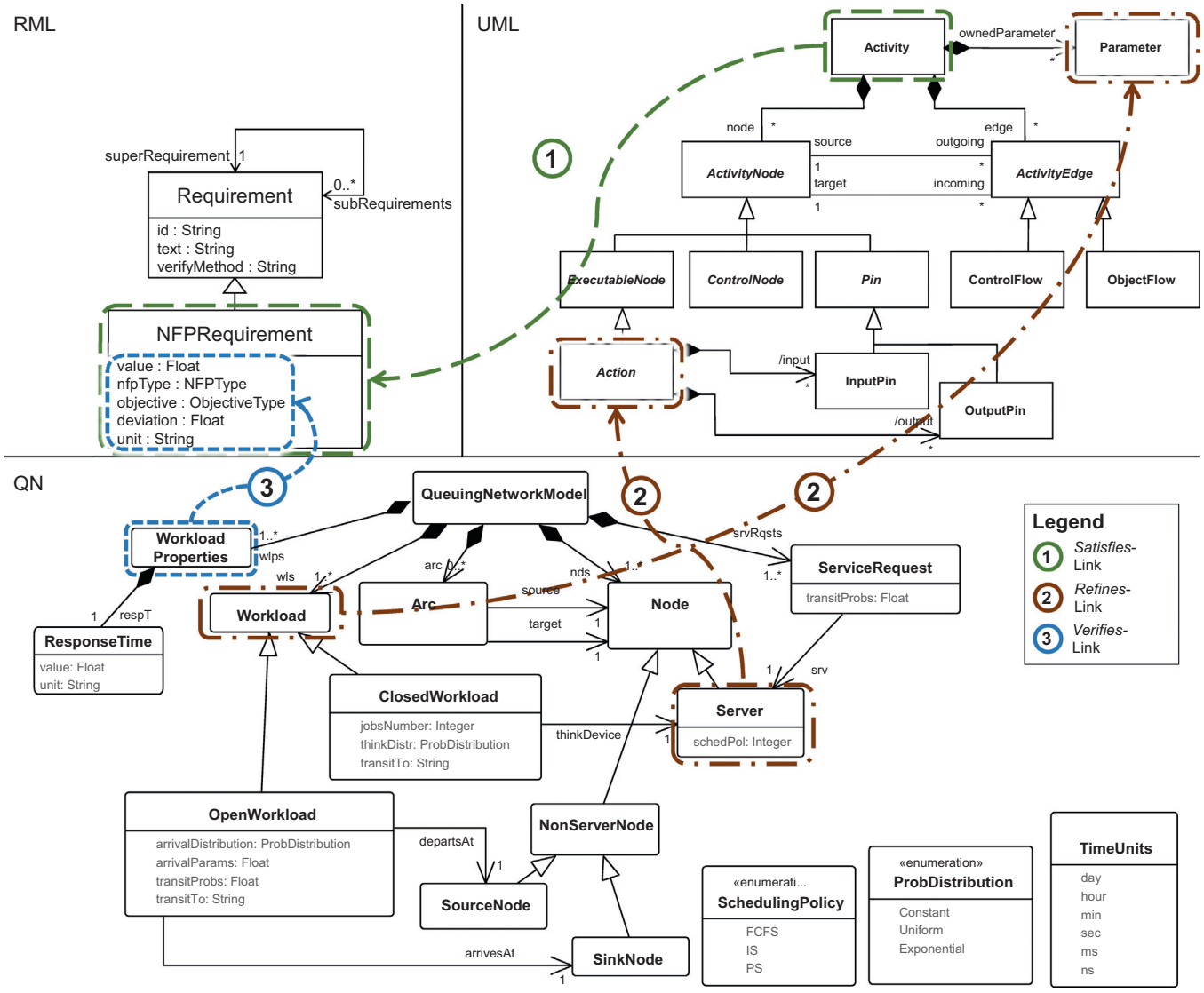


Fig. D.1. Overview on the metamodels in model set 2.

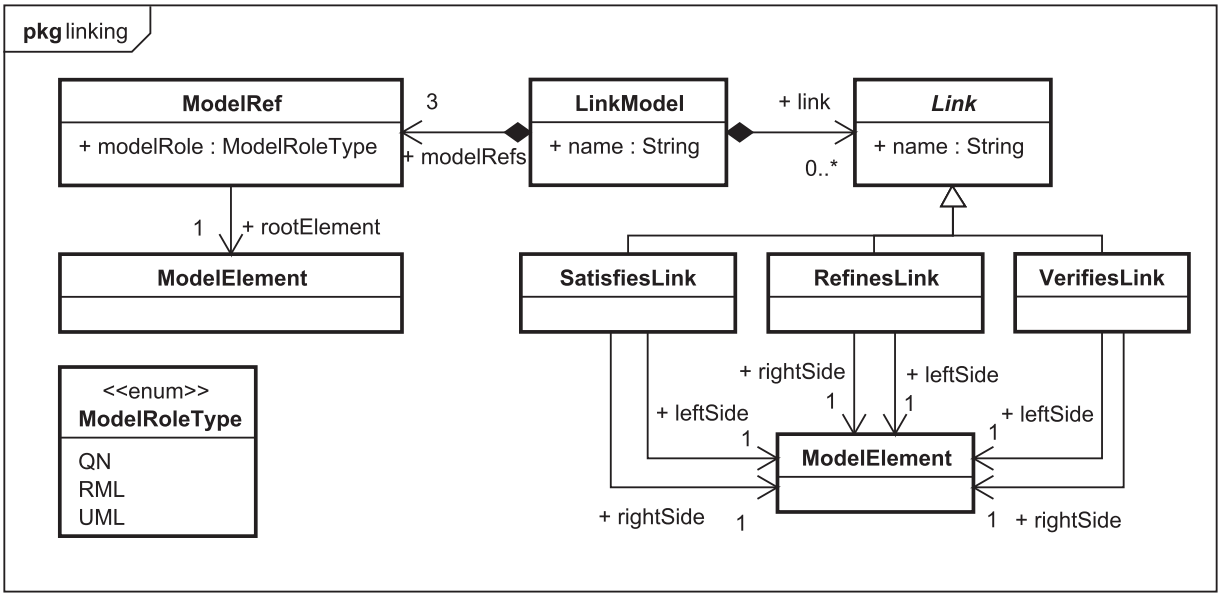


Fig. D.2. Linking language for model set 2 as MOF-based metamodel.

References

Abele, L., Legat, C., Grimm, S., Müller, A.W., 2013. Ontology-based validation of plant models. In: Proceedings of the 11th IEEE International Conference on Industrial Informatics (INDIN), pp. 236–241.

Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y., 2006. Model traceability. IBM Syst. J. – Model-Driven Softw.Dev. 45 (3), 515–526.

Anjorin, A., Rose, S., Deckwerth, F., Schürr, A., 2014. Efficient model synchronization with view triple graph grammars. In: Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA). Springer, pp. 1–17.

Bangor, A., Kortum, P.T., Miller, J.T., 2008. An empirical evaluation of the system usability scale. Int. J. Hum. Comput. Interact. 24 (6), 574–594.

Barbieri, G., Fantuzzi, C., Borsari, R., 2014. A model-based design methodology for the development of mechatronic systems. Mechatronics 24 (7), 833–843.

Bassi, L., Secchi, C., Bonfé, M., Fantuzzi, C., 2011. A SysML-based methodology for manufacturing machinery modeling and design. IEEE/ASME Trans. Mechatron. 16 (6), 1049–1062.

Berardinelli, L., Mazak, A., Alt, O., Wimmer, M., Kappel, G., 2016. Model-driven systems engineering: principles and application in the cpps domain. In: Multi-Disciplinary Engineering for CPPS. Springer, pp. 1–40.

Bergmann, G., Ráth, I., Varró, G., Varró, D., 2012. Change-driven model transformations – change (in) the rule to rule the change. Softw. Syst. Model. 11 (3), 431–461.

Bézivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M., Jouault, F., Kolovos, D.S., Kurtev, I., Paige, R.F., 2006a. A canonical scheme for model composition. In: Proceedings of the 2nd European Conference on Model Driven Architecture (ECMDA), pp. 346–360.

Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A., 2006b. Model transformations? Transformation models!. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). Springer, pp. 440–453.

Biffl, S., Kovalenko, O., Lüder, A., Schmidt, N., Rosendahl, R., 2014. Semantic mapping support in automationml. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), pp. 1–4.

Bill, R., Gogolla, M., Wimmer, M., 2016. On leveraging UML/OCL for model synchronization. In: Proceedings of the 10th Workshop on Models and Evolution Co-Located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), pp. 20–29.

Brambilla, M., Cabot, J., Wimmer, M., 2012. Model-Driven Software Engineering in Practice. Morgan & Claypool.

Brooke, J., 1996. SUS: a "quick and dirty" usability scale. Usability Evaluation in Industry. Taylor and Francis.

Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D., 2010. Seamless model-based development: from isolated tools to integrated model engineering environments. Proc. IEEE 98 (4), 526–545.

Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A., 2011. JTL: a bidirectional and change propagating transformation language. In: Proceedings of the International Conference on Software Language Engineering (SLE). In: Lecture Notes in Computer Science, 6563. Springer, pp. 183–202.

Dávid, I., Meyers, B., Vanherpen, K., Tendeloo, Y.V., Berx, K., Vangheluwe, H., 2017. Modeling and enactment support for early detection of inconsistencies in engineering processes. 2nd International Workshop on Collaborative Modelling in MDE.

Dávid, I., Syriani, E., Verbrugge, C., Buchs, D., Blouin, D., Cicchetti, A., Vanherpen, K., 2016. Towards inconsistency tolerance by quantification of semantic inconsistencies. 1st International Workshop on Collaborative Modelling in MDE.

Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J., 2008. Engineering a DSL for software traceability. In: Proceedings of the 1st International Conference on Software Language Engineering (SLE), pp. 151–167.

Eclipse Foundation, 2015. Papyrus modeling environment. Online. https://eclipse.org/papyrus/.

Egyed, A., 2011. Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. 37 (2), 188–204.

Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A., 2008. Change management in multi-viewpoint system using ASP. In: Workshops Proceedings of the International IEEE Enterprise Distributed Object Computing Conference, pp. 433–440.

Eramo, R., Pierantonio, A., Rosa, G., 2015. Managing uncertainty in bidirectional model transformations. In: Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE), pp. 49–58.

Feldmann, S., Fuchs, J., Vogel-Heuser, B., 2012. Modularity, variant and version management in plant automation – future challenges and state of the art. In: Proceedings of the International Design Conference, pp. 1689–1698.

Feldmann, S., Herzig, S.J., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J., Vogel-Heuser, B., 2015a. Towards effective management of inconsistencies in model-based engineering of automated production systems. IFAC-PapersOnLine 48 (3), 916–923.

Feldmann, S., Herzig, S.J.I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J.J., Vogel-Heuser, B., 2015b. A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study. In: Proceedings of the 11th IEEE International Conference on Autom. Sci. Eng., pp. 158–165.

Feldmann, S., Kernschmidt, K., Vogel-Heuser, B., 2014. Combining a SysML-based modeling approach and semantic technologies for analyzing change influences in manufacturing plant models. In: Proceedings of the CIRP Conference on Manufacturing Systems, pp. 451–456.

Feldmann, S., Kernschmidt, K., Vogel-Heuser, B., 2016a. Applications of semantic web technologies for the engineering of automated production systems – three use cases. In: Biffl, S., Sabou, M. (Eds.), Semantic Web Technologies in Intelligent Engineering Applications. Springer, pp. 353–382.

Feldmann, S., Wimmer, M., Kernschmidt, K., Vogel-Heuser, B., 2016b. A comprehensive approach for managing inter-model inconsistencies in automated production systems engineering. In: Proceedings of the IEEE International Conference on Automation Science and Engineering.

Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., 1994. Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. 20 (8), 569–578.

Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J., 2009. Management of cross-domain model consistency during the development of advanced mechatronic systems. In: Proceedings of the 17th International Conference on Engineering Design, pp. 1–12.

Gero, J.S., 1990. Design prototypes: a knowledge representation schema for design. AI Mag. 11 (4), 26–36.

Giese, H., Wagner, R., 2009. From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. 8 (1), 21–43.

Gnaho, C., Semmak, F., Laleau, R., 2013. An overview of a SysML extension for goal-oriented nfr modelling. In: Proceedings of the IEEE 7th International Conference on Research Challenges in Information Science (RCIS), pp. 1–2.

Göring, M., Fay, A., 2012. Modeling change and structural dependencies of automation systems. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012).

Guerra, E., de Lara, J., Orejas, F., 2013a. Inter-modelling with patterns. Softw. Syst. Model. 12 (1), 145–174.

Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., 2013b. Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. 20 (1), 5–46.

Hegedüs, A., Horváth, A., Ráth, I., Branco, M.C., Varró, D., 2011. Quick fix generation for dsmls. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 17–24.

Hehenberger, P., Egyed, A., Zeman, K., 2010. Consistency checking of mechatronic design models. In: Proceedings of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, pp. 1141–1148.

Hermann, F., Ehrig, H., Ermel, C., Orejas, F., 2012. Concurrent model synchronization with conflict resolution based on triple graph grammars. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). In: Lecture Notes in Computer Science, 7212. Springer, pp. 178–193.

Herzig, S., Qamar, A., Reichwein, A., Paredis, C., 2011. A conceptual framework for consistency management in model-based systems engineering. In: ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference. Washington, DC, USA, pp. 1329–1339. doi:10.1115/DETC2011-47924.

Herzig, S.J., Qamar, A., Paredis, C.J., 2014. An approach to identifying inconsistencies in model-based systems engineering. In: Proceedings of the Conference on Systems Engineering Research, pp. 354–362.

Hidaka, S., Tisi, M., Cabot, J., Hu, Z., 2016. Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. 15 (3), 907–928.

Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L., 2005. Consistency problems in uml-based software development. In: UML Modeling Languages and Applications. In: Lecture Notes in Computer Science, 3297. Springer, pp. 1–12.

International Electrotechnical Commission, 2009. Iec 81346-2:2009 – industrial systems, installations and equipment and industrial products – structuring principles and reference designations – part 2: Classification of objects and codes for classes.

Jäger, T., Fay, A., Wagner, T., Löwen, U., 2012. Comparison of engineering results within domain specific languages regarding information contents and intersections. In: Proceedings of the International Multi-Conference on Systems, Signals and Devices, pp. 1–6.

Jossic, A., Fabro, M.D.D., Lerat, J., Bézivin, J., Jouault, F., 2007. Model integration with model weaving: a case study in system architecture. In: Proceedings of the 1st IEEE International Conference on Systems Engineering and Modeling, pp. 79–84.

Kammerl, D., Malaschewski, O., Schenkl, S.A., Mörtl, M., 2015. Decision uncertainties in the planning of product-service system portfolios. In: Proceedings of the 5th International Conference Conference on Research into Design, pp. 39–48.

Kindler, E., Wagner, R., 2007. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Tech. Rep. tr-ri-07-284. University of Paderborn.

Ko, A.J., Latoza, T.D., Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empir. Softw. Eng. 20 (1), 110–141. doi:10.1007/s10664-013-9279-3.

Kolovos, D.S., Paige, R.F., Polack, F., 2006. Merging models with the Epsilon Merging Language (EML). In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, pp. 215–229.

Kovalenko, O., Serral, E., Sabou, M., Ekaputra, F.J., Winkler, D., Biffl, S., 2014. Automating cross-disciplinary defect detection in multi-disciplinary engineering environments. In: Proceedings of the International Conference on Knowledge Engineering and Knowledge Management, pp. 238–249.

Kühne, T., 2006. Matters of (meta-)modeling. Softw. Syst. Model. 5 (4), 369–385.

Lange, C., Chaudron, M., 2004. An empirical assessment of completeness in UML designs. In: Proceedings of International Conference on Empirical Assessment in Software Engineering, pp. 111–121.

Langer, P., Wieland, K., Wimmer, M., Cabot, J., 2012. EMF Profiles: a lightweight extension approach for EMF models. J. Object Technol. 11 (1), 1–29.
Macedo, N., Cunha, A., 2014. Least-change bidirectional model transformation with QVT-R and ATL. Softw. Syst. Model. 1–28.
Maro, S., Steghöfer, J., 2016. Capra: a configurable and extendable traceability management tool. In: 24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12–16, 2016, pp. 407–408.
Mens, T., Van Der Straeten, R., D'Hondt, M., 2006. Detecting and resolving model inconsistencies using transformation dependency analysis. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems, pp. 200–214.
Mens, T., Van Der Straeten, R., Simmonds, J., 2005. A framework for managing consistency of evolving UML models. In: Software Evolution with UML and XML. Idea Group Publishing, pp. 1–30.
Moser, T., Biffl, S., 2012. Semantic integration of software and systems engineering environments. IEEE Trans. Syst. ManCybern. Part C 42 (1), 38–50.
Muskens, J., Bril, R.J., Chaudron, M.R.V., 2005. Generalizing consistency checking between software views. In: Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005), 6–10 November 2005, Pittsburgh, Pennsylvania, USA, pp. 169–180.
NoMagic Inc., 2016. Magicdraw. Online. http://www.nomagic.com/products/magicdraw.html.
Nuseibeh, B., Easterbrook, S., Russo, A., 2000. Leveraging inconsistency in software development. Computer 33 (4), 24–29.
Object Management Group, 2015a. Systems modeling language version 1.4. Online. http://www.omg.org/spec/SysML/1.4/.
Object Management Group, 2015b. Unified modeling language version 2.5. Online. http://www.omg.org/spec/UML/2.5/.
Object Management Group (OMG), 2003. Meta Object Facility (MOF) 2.0 Core Specification. Online. http://www.omg.org/spec/MOF/2.0/.
Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131–164.
Schäfer, W., Wehrheim, H., 2010. Model-driven development with mechatronic uml. In: Graph Transformations and Model-Driven Engineering. Springer, pp. 533–554.
Schätz, B., Braun, P., Huber, F., Wisspeintner, A., 2003. Consistency in model-based development. In: Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, pp. 287–296.
Schmidt, D., 2006. Guest editor's introduction: model-driven engineering. Computer 39 (2), 25–31.
Schoenboeck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M., 2014. CARE: a constraint-based approach for re-establishing conformance relationships. In: Proceedings of the 10th Asia-Pacific Conference on Conceptual Modelling (APCCM). Australian Computer Society, pp. 19–28.
Schürr, A., 1994. Specification of graph translators with triple graph grammars. In: Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG). Springer, pp. 151–163.
Schürr, A., Klar, F., 2008. 15 years of triple graph grammars. In: Proceedings of the 4th International Conference on Graph Transformation, pp. 411–425.
Spanoudakis, G., Zisman, A., 2001. Inconsistency management in software engineering: survey and open research issues. In: Handbook of Software Engineering & Knowledge Engineering: Fundamentals. World Scientific Publishing, pp. 329–380.
Stevens, P., 2009. Bidirectional model transformations in QVT: semantic issues and open questions. Softw. Syst. Model. 9 (1), 7–20.
Stevens, P., 2017. Bidirectional transformations in the large. In: 20th International Conference on Model Driven Engineering Languages and Systems, pp. 1–11.
Thramboulidis, K., 2013. Overcoming mechatronic design challenges: the 3+1 SysML-view model. Comput. Sci. Technol. Int. J. 1 (1), 6–14.
Troya, J., Vallecillo, A., 2014. Specification and simulation of queuing network models using domain-specific languages. Comput. Stand. Interfaces 36 (5), 863–879.
Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V., 2003. Using description logic to maintain consistency between UML models. In: Proceedings of the International Conference on the Unified Modeling Language (UML). In: Lecture Notes in Computer Science, 2863. Springer, pp. 326–340.
Vogel-Heuser, B., 2015. Usability experiments to evaluate uml/sysml-based model driven software engineering notations for logic control in manufacturing automation. J. Softw. Eng. Appl. 11 (7), 943–973.
Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M., 2015. Evolution of software in automated production systems: challenges and research directions. J. Syst. Softw. 110, 54–84.
Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S., 2014. Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. Tech. Rep. TUM-AIS-TR-01-14-02. Technical University of Munich. Online. https://mediatum.ub.tum.de/node?id=1208973.
Vogel-Heuser, B., Rösch, S., 2015. Applicability of technical debt as a concept to understand obstacles for evolution of automated production systems. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 127–132.
Wirth, N., 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM 20 (11), 822–823.
Zolotas, A., Rodriguez, H.H., Kolovos, D.S., Paige, R.F., Hutchesson, S., 2017. Bridging proprietary modelling and open-source model management tools: the case of ptc integrity modeller and epsilon. In: International Conference on Model Driven Engineering Languages and Systems, pp. 237–247. doi:10.1109/MODELS.2017.18.
Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A., 2009. Domain-specific metamodelling languages for software language engineering. In: Proceedings of the 2nd International Conference on Software Language Engineering (SLE), pp. 334–353.

Stefan Feldmann received the Dipl.-Ing. degree in mechanical engineering from Technische Universität München, Germany, in 2012. He is researching towards the PhD degree at the Institute of Automation and Information Systems, Technische Universität München. His current research interests focus on the application of Semantic Web Technologies for improving interdisciplinary engineering in the domain of production systems.

Konstantin Kernschmidt received the Dipl.-Ing. degree in mechanical engineering from Technische Universität München, Germany, in 2011. He is researching towards the PhD degree at the Institute of Automation and Information Systems, Technische Universität München. His current research interests focus on the interdisciplinary model based engineering in the field of mechatronic production systems.

Manuel Wimmer received his PhD degree in business informatics in 2008 and his Habilitation (venia docendi) in informatics in 2014, both from the Technische Universität Wien. In 2016, he has been a visiting professor in the Institute of Automation and Information Systems at the Technische Universität München. Currently, he is a full professor leading the Institute of Business Informatics - Software Engineering at the Johannes Kepler University Linz. His current research interests are focused on the foundations and applications of modeling technologies.

Birgit Vogel-Heuser graduated in electrical engineering and received the PhD in mechanical engineering from the RWTH Aachen in 1991. She worked for nearly ten years in industrial automation in the machine and plant manufacturing industry. After holding different chairs of automation she has since 2009 been head of the Institute of Automation and Information Systems at the Technische Universität München. Her research work is focused on modeling and education in automation engineering for distributed and intelligent systems.