Technische Universität München
Lehrstuhl für Automatisierung und Informationssysteme

Diagnosis and Handling of Inconsistencies in Heterogeneous Models of Automated Production Systems

Dipl.-Ing. (Univ.) Stefan Feldmann

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Markus Zimmermann
Prüfende der Dissertation: 1. Prof. Dr.-Ing. Birgit Vogel-Heuser
2. Prof. Dr. Helmut Krcmar
3. Prof. Dr.techn. Gertrude Kappel

Die Dissertation wurde am 13.06.2019 bei der Technische Universität München eingereicht und durch die Fakultät für Maschinenwesen am 05.09.2019 angenommen.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Diagnosis and Handling of Inconsistencies in Heterogeneous Models of Automated Production Systems

Autor:
Stefan Feldmann

ISBN 13: 978-3-96548-053-7 (Print)
ISBN 13: 978-3-96548-054-4 (E-Book)

1. Auflage 2019

Copyright
© sierke VERLAG
Friedländer Weg 54
37085 Göttingen
Tel.: +49 (0)551 5036647

Coverdesign: sierke MEDIA

Acknowledgements

Writing and finalizing a dissertation would not have been possible without the tremendous support of great people.

First of all, I would like to thank Professor Birgit Vogel-Heuser for her support, guidance and understanding during the last years. I would also like to thank the members of my doctoral committee, Professor Gertrude Kappel and Professor Helmut Krcmar, for investing the effort for finalizing this dissertation. In addition, I would like to thank the Deutsche Forschungsgemeinschaft (DFG) for the financial support within the Collaborative Research Centre (CRC) 768 "Managing Cycles in Innovation Processes – Integrated Development of Product-Service Systems Based on Technical Products".

I would also like to thank the colleagues and members of the Institute of Automation and Information Systems for their friendship, fruitful discussions and support. In particular, I would like to thank Susanne Rösch, Konstantin Kernschmidt, Sebastian Rehberger, Daniel Schütz, Jens Folmer, Dorothea Pantförder, Gennadiy Koltun and Felix Ocker.

In addition, both researchers and industry partners with whom I had the pleasure to work with during the course of joint research, workshops, conferences or projects shaped the results of this dissertation. In particular, I would like to thank Sebastian J.I. Herzig and Ahsan Qamar as well as their adviser, Professor Christiaan J.J. Paredis. Moreover, I would like to thank Professor Manuel Wimmer. In a similar manner, I would like to thank Sebastian Diehm and Michael Schwarz from Schneider Electric Automation GmbH for great discussions regarding practical applications and implementations.

Finally, I would like to thank my family. Thank you for all your love and trust – thank you for always believing in me – thank you for everything.

Abstract

Companies in the automated production systems domain are more and more forced to manufacture individualized goods in order to remain globally competitive. Thereby, the rapidly changing product and system requirements impose an ever-increasing complexity on the engineering of automated production systems. One opportunity to reduce this complexity and to support engineers in developing automated production systems is the use of models. However, due to the manifold disciplines involved in the engineering process – mechanical, electrical/ electronic and software engineering to name only a few – and the disparate stakeholders participating in this process, manifold heterogeneous models are created during engineering. To make things worse, these models heavily differ – for instance, in their degree of formality, with regard to their level of abstraction or regarding their viewpoint on the system. In addition, the models overlap – they incorporate elements, which refer to common aspects of the system under investigation. As a consequence, inconsistencies – namely states of conflicts within the models – are likely to occur. Whereas in the current state of practice, a number of commercially available software tools already provide the means for checking the compliance to syntactical or well-formedness constraints, none of these tools support the overarching management of inconsistencies within engineering models of automated production systems. To address these issues, manifold approaches have been developed in the related research, which focus on the diagnosis of inconsistencies but not on the handling of inconsistencies in engineering models of automated production systems. It is therefore inevitable to provide an approach, which allows to diagnose inconsistencies within the multitude of models created during engineering and, in case an inconsistency is diagnosed, to support engineers in choosing appropriate handling actions.

In this dissertation, based on the different heterogeneous types of models and inconsistencies in the automated production systems domain, requirements to be fulfilled by an inconsistency management approach are derived. Furthermore, a knowledge-based approach to diagnose and handle inconsistencies by means of Semantic Web technologies – in particular the Resource Description Framework (RDF) and the SPARQL Protocol and RDF Query Language (SPARQL) – is presented. Inconsistency diagnosis rules are therein described by means of graph patterns (SPARQL Query Language), which are matched against a graph-based representational formalism (RDF triples) that contains the involved models. Analogously, inconsistency handling rules are described by means of respective graph manipulation rules (SPARQL Update Language) and executed to handle diagnosed inconsistencies by either ignoring, tolerating or resolving them. To keep the inconsistency management approach as flexible as possible, semantic abstraction mechanisms are provided by means of mediations to vocabularies, which allow for incorporating not only the different disciplines involved during engineering, but also discipline-spanning concepts and background knowledge that are common to multiple disciplines. Hence, instead of creating a "world model" of automated production systems, models are mediated to a common model that represents only the elements that are required for inconsistency management purposes.

For the purpose of validating the approach, a prototypical implementation is realized by means of standard implementations within the fields of Model-based Engineering (Eclipse Modeling Framework (EMF)) and Semantic Web technologies (Apache Jena). The overall feasibility of the approach is evaluated by means of a laboratory automated production system. In addition, the approach is compared to another inconsistency management approach, which relies on Model-based Engineering technologies (EMF). Finally, in order to assess benefits and limitations of the approach, an evaluation at the hand of application examples that are inspired from industrial application is provided.

Zusammenfassung

Um global wettbewerbsfähig zu bleiben, müssen Unternehmen im Bereich automatisierter Produktionssysteme immer mehr individualisierte Güter fertigen können. Die sich schnell ändernden Produkt- und Systemanforderungen stellen dabei immer komplexere Anforderungen an das Engineering automatisierter Produktionssysteme. Eine Möglichkeit, diese Komplexität zu beherrschen und Ingenieure bei der Entwicklung automatisierter Produktionssysteme zu unterstützen, ist der Einsatz von Modellen. Aufgrund der vielfältigen Disziplinen des Engineering-Prozesses – beispielsweise mechanische, elektrische/elektronische und Software-Entwicklung – und der unterschiedlichen Akteurinnen und Akteure, die an diesem Prozess beteiligt sind, entstehen während des Engineerings vielfältige heterogene Modelle. Erschwerend kommt hinzu, dass sich diese Modelle stark unterscheiden – zum Beispiel in ihrem Formalitätsgrad, in Bezug auf ihren Abstraktionsgrad oder in Bezug auf ihre Sicht auf das System. Darüber hinaus überlappen sich die Modelle – sie enthalten Elemente, die sich auf gemeinsame Aspekte des untersuchten Systems beziehen. Infolgedessen ist es wahrscheinlich, dass Inkonsistenzen – d.h. Konfliktzustände innerhalb der Modelle – auftreten. Im aktuellen Stand der Technik existieren zwar bereits eine Reihe kommerziell erhältlicher Software-Tools, die syntaktische oder Wohlgeformtheits-Regeln überprüfen können; keines dieser Tools unterstützt jedoch das übergreifende Management solcher Inkonsistenzen innerhalb von Engineering-Modellen automatisierter Produktionssysteme. Zur Adressierung der Problematik wurden bereits einige Ansätze entwickelt, die sich auf die Diagnose von Inkonsistenzen konzentrieren, jedoch nicht auf die Handhabung in den Modellen automatisierter Produktionssysteme. Daher ist es essenziell, Ingenieurinnen und Ingenieure bei der Diagnose und Handhabung von Inkonsistenzen zu unterstützen.

In dieser Dissertation werden zunächst basierend auf den unterschiedlichen heterogenen Modellen und Inkonsistenzen im Bereich der automatisierten Produktionssysteme Anforderungen abgeleitet, die von einem Inkonsistenzmanagement-Ansatz erfüllt werden müssen. Darüber hinaus wird ein wissensbasierter Ansatz zur Diagnose und Handhabung von Inkonsistenzen mit Hilfe von Semantic Web-Technologien vorgestellt – insbesondere mittels RDF und SPARQL. Regeln zur Diagnose von Inkonsistenzen werden darin mittels Graph-Muster (SPARQL Query Language) beschrieben, die gegen einen graphenbasierten Repräsentationsformalismus ausgeführt werden (RDF-Tripel), welcher die jeweiligen Modelle abbildet. Analog dazu werden Regeln zur Handhabung von Inkonsistenzen mit Hilfe entsprechender Regeln zur Manipulation der Graphen (SPARQL Update Language) formuliert und ausgeführt, um diagnostizierte Inkonsistenzen zu handhaben, indem sie entweder ignoriert, toleriert oder behoben werden. Darüber hinaus werden semantische Abstraktionsmechanismen durch Mediation eingesetzt, um den Inkonsistenzmanagement-Ansatz so flexibel wir möglich zu halten – beispielsweise durch die Abbildung fachübergreifender Konzepte und Hintergrundwissen. Um hierbei kein "Weltmodell" zu erstellen, werden die Modelle auf eine gemeinsame, semantische Ebene gehoben, welche nur die Elemente beinhaltet, die für das Inkonsistenzmanagement erforderlich sind.

Zur Validierung des Ansatzes wird eine prototypische Implementierung mittels Standardimplementierungen in den Bereichen Model-based Engineering (EMF) und Semantic Web Technologies (Apache Jena) bereitgestellt. Die Anwendbarkeit des Ansatzes wird anhand der Modelle eines exemplarischen Produktionssystems bewertet. Darüber hinaus wird der Ansatz mit einem anderen Inkonsistenzmanagement-Ansatz verglichen, der ausschließlich auf modellbasierten Engineering-Technologien basiert (EMF). Abschließend wird eine Bewertung anhand von Anwendungsbeispielen, die sich an der industriellen Anwendung orientieren, vorgenommen, um die Vor- und Nachteile des Ansatzes abzuschätzen.

List of published publications in the context of this dissertation

Journal Articles

Feldmann, S., Kernschmidt, K., and Vogel-Heuser, B. “Konzept eines wissensbasierten Frameworks zur Spezifikation und Diagnose von Inkonsistenzen in mechatronischen Modellen”. German. In: *at – Automatisierungstechnik*, vol. 64, no. 3 (2016), pp. 199–215. DOI: 10.1515/auto-2015-0081.

Feldmann, S., Kernschmidt, K., Wimmer, M., and Vogel-Heuser, B. “Managing Inter-Model Inconsistencies in Model-based Systems Engineering: Application in Automated Production Systems Engineering”. In: *Journal of Systems and Software*, vol. 153 (2019), pp. 105–134. DOI: 10.1016/j.jss.2019.03.060.

Feldmann, S. and Vogel-Heuser, B. “Interdisciplinary Product Lines to Support the Engineering in the Machine Manufacturing Domain”. In: *International Journal of Production Research* (2016). DOI: 10.1080/00207543.2016.1211343.

Vogel-Heuser, B., Fuchs, J., Feldmann, S., and Legat, C. “Interdisziplinärer Produktlinienansatz zur Steigerung der Wiederverwendung”. German. In: *at – Automatisierungstechnik*, vol. 63, no. 2 (2015), pp. 99–110. DOI: 10.1515/auto-2014-1140.

Conference Papers

Feldmann, S., Hauer, F., Ulewicz, S., and Vogel-Heuser, B. “Analysis Framework for Evaluating PLC Software: An Application of Semantic Web Technologies”. In: *IEEE International Symposium on Industrial Electronics*. Santa Clara, CA, USA, 2016.

Feldmann, S., Wimmer, M., Kernschmidt, K., and Vogel-Heuser, B. “A Comprehensive Approach for Managing Inter-Model Inconsistencies in Automated Production Systems Engineering”. In: *IEEE International Conference on Automation Science and Engineering*. Fort Worth, TX, USA, 2016.

Feldmann, S., Fuchs, J., and Vogel-Heuser, B. “Modularity, Variant and Version Management in Plant Automation – Future Challenges and State of the Art”. In: *International Design Conference*. Dubrovnik, Croatia, 2012, pp. 1689–1698. URL: https://www.designsociety.org/publication/32138/modularity_variant_and_version_management_in_plant_automation_%E2%80%93_future_challenges_and_state_of_the_art (visited on 05/20/2019).

Feldmann, S., Hauer, F., Pantförder, D., Pankratz, F., Klinker, G., and Vogel-Heuser, B. “Management of Inconsistencies in Domain-Spanning Models – An Interactive Visualization Approach”. In: *19th International Conference on Human-Computer Interaction*. Vancouver, Canada, 2017.

Feldmann, S., Herzig, S. J. I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C. J. J., and Vogel-Heuser, B. "A Comparison of Inconsistency Management Approaches Using a Mechatronic Manufacturing System Design Case Study". In: *IEEE International Conference on Automation Science and Engineering.* Gothenburg, Sweden, 2015, pp. 158–165. DOI: 10.1109/CoASE.2015.7294055.

Feldmann, S., Herzig, S. J. I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C. J. J., and Vogel-Heuser, B. "Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems". In: *15th IFAC Symposium on Information Control Problems in Manufacturing.* Ottawa, Canada, 2015, pp. 916–923. DOI: 10.1016/j.ifacol.2015.06.200.

Feldmann, S., Kernschmidt, K., and Vogel-Heuser, B. "Combining a SysML-based Modeling Approach and Semantic Technologies for Analyzing Change Influences in Manufacturing Plant Models". In: *CIRP Conference on Manufacturing Systems.* Ontario, Canada, 2014, pp. 451–456. DOI: 10.1016/j.procir.2014.01.140.

Feldmann, S., Legat, C., Kernschmidt, K., and Vogel-Heuser, B. "Compatibility and Coalition Formation: Towards the Vision of an Automatic Synthesis of Manufacturing System Designs". In: *IEEE International Symposium on Industrial Electronics.* Istanbul, Turkey, 2014, pp. 1712–1717. DOI: 10.1109/ISIE.2014.6864873.

Feldmann, S., Legat, C., Schütz, D., Ulewicz, S., and Vogel-Heuser, B. "Automatic Rule-Based Inference of Control Software Capabilities Considering Interdisciplinary Aspects". In: *International Conference on Production Research.* Iguassu Falls, Brazil, 2013.

Feldmann, S., Legat, C., and Vogel-Heuser, B. "An Analysis of Challenges and State of the Art for Modular Engineering in the Machine and Plant Manufacturing Domain". In: *IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control.* Maribor, Slovenia, 2015, pp. 87–92. DOI: 10.1016/j.ifacol.2015.08.113.

Feldmann, S., Legat, C., and Vogel-Heuser, B. "Engineering Support in the Machine Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis". In: *IFAC Symposium on Information Control Problems in Manufacturing.* Ottawa, Canada, 2015, pp. 211–218. DOI: 10.1016/j.ifacol.2015.06.083.

Feldmann, S., Loskyll, M., Rösch, S., Schlick, J., Zühlke, D., and Vogel-Heuser, B. "Increasing Agility in Engineering and Runtime of Automated Manufacturing Systems". In: *IEEE International Conference on Industrial Technology.* Cape Town, South Africa, 2013, pp. 1303–1308. DOI: 10.1109/ICIT.2013.6505861.

Feldmann, S., Rösch, S., Legat, C., and Vogel-Heuser, B. "Keeping Requirements and Test Cases Consistent: Towards an Ontology-based Approach". In: *IEEE International Conference on Industrial Informatics.* Porto Alegre, Brazil, 2014, pp. 726–732. DOI: 10.1109/INDIN.2014.6945603.

Feldmann, S., Rösch, S., Schütz, D., and Vogel-Heuser, B. "Model-Driven Engineering and Semantic Technologies for the Design of Cyber-Physical Systems". In: *IFAC Workshop on Intelligent Manufacturing Systems.* São Paulo, Brazil, 2013, pp. 210–215. DOI: 10.3182/20130522-3-BR-4036.00050.

Fuchs, J., Feldmann, S., and Vogel-Heuser, B. "Modularität im Maschinen- und Anlagenbau – Analyse der Anforderungen und Herausforderungen im industriellen Einsatz". German. In: *Entwurf komplexer Automatisierungssysteme.* Magdeburg, Germany, 2012, pp. 307–316.

Fuchs, J., Feldmann, S., Legat, C., and Vogel-Heuser, B. "Identification of Design Patterns for IEC 61131-3 in Machine and Plant Manufacturing". In: *IFAC World Congress.* Cape Town, South Africa, 2014, pp. 6092–6097. DOI: 10.3182/20140824-6-ZA-1003.01595.

Kellner, A., Weingartner, L., Friedl, M., Hehenberger, P., Ahrens, M., Kernschmidt, K., Feldmann, S., Vogel-Heuser, B., and Zeman, K. "Challenges in Integrating Requirements in Model Based Development Processes in the Machinery and Plant Building Industry". In: *IEEE International Symposium on Systems Engineering.* Edinburgh, Scotland, 2016.

Koltun, G. D., Feldmann, S., Schütz, D., and Vogel-Heuser, B. "Model-Document Coupling in aPS Engineering: Challenges and Requirements Engineering Use Case". In: *18th IEEE International Conference on Industrial Technology.* 2017. DOI: 10.1109/ICIT.2017.7915529.

Legat, C., Seitz, C., Lamparter, S., and Feldmann, S. "Semantics to the Shop Floor: Towards Ontology Modularization and Reuse in the Automation Domain". In: *IFAC World Congress.* Cape Town, South Africa, 2014, pp. 3444–3449. DOI: 10.3182/20140824-6-ZA-1003.02512.

Legat, C., Steden, F., Feldmann, S., Weyrich, M., and Vogel-Heuser, B. "Co-evolution and Reuse of Automation Control and Simulation Software: Identification and Definition of Modification Actions and Strategies". In: *Annual Conference of the IEEE Industrial Electronics Society.* Dallas, TX, USA, 2014, pp. 2525–2531. DOI: 10.1109/IECON.2014.7048861.

Vogel-Heuser, B., Fischer, J., Rösch, S., Feldmann, S., and Ulewicz, S. "Challenges for Maintenance of PLC-Software and its Related Hardware for Automated Production Systems: Selected Industrial Case Studies". In: *IEEE International Conference on Software Maintenance and Evolution.* Bremen, Germany, 2015, pp. 362–371. DOI: 10.1109/ICSM.2015.7332487.

Book Sections

Feldmann, S., Kernschmidt, K., and Vogel-Heuser, B. "Applications of Semantic Web Technologies for the Engineering of Automated Production Systems – Three Use Cases". In: *Semantic Web Technologies in Intelligent Engineering Applications.* Ed. by **Biffl, S. and Sabou, M.** Berlin, Heidelberg, Germany: Springer, 2016.

Feldmann, S. and Vogel-Heuser, B. "Diagnose von Inkonsistenzen in heterogenen Engineeringdaten". German. In: *Industrie 4.0 in Produktion, Automatisierung und Logistik.* Ed. by **Bauernhansl, T., ten Hompel, M., and Vogel-Heuser, B.** Berlin, Heidelberg, Germany: Springer, 2016, pp. 1–21. DOI: 10.1007/978-3-662-45537-1_91-1.

Contents

Chapter 1.

Introduction

This dissertation is motivated by the ever-increasing complexity of the engineering process in the automated production systems domain. In particular, it is aimed at providing a framework that supports engineers in better diagnosing and handling inconsistencies that are likely to occur during such an interdisciplinary process. Hence, the need for inconsistency management in heterogeneous models in the automated production systems domain is motivated further in this chapter. Specifically, Section 1.1 details the context and motivation of this dissertation. In Section 1.2, the research objectives are presented. Finally, in Section 1.3, the outline of the dissertation is introduced.

1.1. Context and Motivation

Industrial production processes have been and are currently evolving, especially due to the increasing need to rapidly react to changes in the product or system requirements [LCK15; ElM06]. These changes mainly result from drivers such as the increasing globalization of companies, the resulting need to compete in global markets, fluctuating market dynamics and from adaptations in the consumer behaviour. Consequently, companies in the industrial production domain are confronted with an increasing complexity of industrial production processes and with the growing need to improve effectiveness and efficiency during engineering and operation of their systems. As a consequence, the engineering of automated production systems increases steadily, making appropriate support during engineering necessary [VH16].

However, the challenges for engineering in the automated production systems domain are manifold: Different stakeholders from diverse disciplines are involved during this process [Vog+15], e.g. from mechanical, electrical and software engineering, but also from additional disciplines such as requirements and project engineering. To address the complexity of automated production systems in an appropriate fashion, the involved stakeholders work in different, interdisciplinary teams and, therefore, use different notational formalisms, abstraction levels and software tools to address their specific concerns [Gau+09; Bro+10]. Hence, a multitude of heterogeneous models is created concurrently. Although these disparate models address different perspectives within the engineering, they are not completely disjoint – instead, they overlap at some points. As a result, it is likely that *inconsistencies*, namely a conflict or contradiction between these models [SZ01], arise. It is essential to cope with these by continuously diagnosing and handling them.

Some inconsistencies can lead to severe consequences. Popular examples in engineering projects are the Mars Climate Orbiter [NAS00], whose mission failed due to a mismatch in the applied unit system, and the Denver International Airport baggage system [Neu94], in which underestimation of the system's complexity and missing coordination in between interdisciplinary teams lead to costly delays in system delivery. Nevertheless, inconsistencies can also have less serious outcomes, which lead to delays or increased costs throughout the engineering of automated production systems. For instance, the commissioning of automated production systems is often a time-critical process, in which design decisions are poorly revised. As a consequence, the personnel on site implements "quick hacks", which may lead to a *technical debt* [VR15] – namely extra development time and costs that arise when refraining from applying the optimal solution. Especially in parallel, interdisciplinary engineering, in which the effects of changing one discipline-specific part of the solution to other disciplines are often not captured explicitly [VR15], it is most likely that inconsistencies and, hence, technical debt arise.

Motivated by this ever-increasing demand of companies to improve their engineering processes, this dissertation aims at providing a framework for managing inconsistencies – that is: for continuously diagnosing and handling inconsistencies throughout the interdisciplinary engineering process of automated production systems. Whereas in the current state of practice, a number of commercially available software tools already provide the means for checking the compliance to syntactical or well-formedness constraints, none of these tools support the overarching management of inconsistencies within engineering models of automated production systems. To address these issues, manifold approaches have been developed in the related research, which focus on the diagnosis of inconsistencies but not on the handling of inconsistencies in engineering models of automated production systems. It is therefore assumed that a flexible, knowledge-based approach is required, which (1) captures the engineering models of automated production systems within a *central knowledge base* and allows to (2) *diagnose inconsistencies* within the multitude of models created during engineering as well as, in case an inconsistency is diagnosed, (3) to support engineers in *choosing appropriate handling actions* to be taken. This assumption is the basic hypothesis, which will be validated throughout the remainder of this dissertation.

1.2. Research Objectives

In order to achieve such an inconsistency management framework, this dissertation answers the overall research questions on how inconsistencies in engineering models of automated production systems can be diagnosed and handled. Therein, it is not aimed at a fully automatic solution to diagnose and handle inconsistencies; rather, it is aimed at supporting the responsible stakeholders, e.g., discipline-specific engineers, to decide, which handling action must be taken in case of an inconsistency. Consequently, this dissertation aims at an interactive inconsistency management framework that should not be seen as a fully automatic tool – but rather as a supporting framework for domain experts in the automated production systems domain. Hence, the central research question answered in this dissertation can be formulated as follows.

Central Research Question — *How can inconsistencies in engineering models of automated production systems be managed (that is: diagnosed and handled) (semi-)automatically?*

From this overall research question, several dedicated research questions can be derived. First, the question arises, which typical strategies for inconsistency management exist in both the related state of practice and research. Consequently, the related research as well as commercially available frameworks that aid domain experts in managing inconsistencies in the automated production systems domain are compared in this dissertation.

Research Question 1 — *What are typical strategies to diagnose and handle inconsistencies? What are the strengths and limitations of current inconsistency diagnosis and handling approaches?*

Second, as a multitude of models is created throughout the interdisciplinary engineering process of automated production systems, the question arises, what the content of engineering models of automated production systems is that needs to be incorporated in inconsistency management. Hence, a set of dedicated metamodels is formulated to explicitly capture the content that stems from the typical engineering disciplines – e.g., from the mechanical, electrical and software engineering disciplines, or from further disciplines such as hydraulics or pneumatics. Therein, it is not focused on specific tools that are typically used for discipline-specific viewpoints of the engineering, but rather on the information content that is created throughout the engineering.

Research Question 2 — *How can the content of typical models in the engineering of the automated production systems domain be captured for the purpose of inconsistency management?*

Third, based on the essential information content created during the interdisciplinary engineering of automated production systems, appropriate mechanisms to diagnose and handle inconsistencies can be put in place. However, the question arises, to what extent typical inconsistencies in engineering models of automated production systems can be diagnosed and handled in a (semi-)automatic manner. Therefore, a framework is presented throughout this dissertation, that allows for diagnosing and handling typical inconsistencies in the engineering of automated production systems domain.

Research Question 3 — *To what extent can typical inconsistencies in engineering models of automated production systems be diagnosed and handled (semi-)automatically by an inconsistency management framework?*

1.3. Outline of the Dissertation

To answer the previously formulated research questions, the remainder of this dissertation is structured as follows. In the next chapter, terms and definitions that are used throughout this dissertation are defined, and the context for the following chapters is given in Chapter 2. Subsequently, the requirements to be fulfilled by an inconsistency management framework for engineering models in the automated production systems domain as well as the simplifying assumptions for this dissertation are derived in Chapter 3. In Chapter 4, related works in research and practice are discussed and compared to the objectives of this dissertation. Chapter 5 introduces the overall concept for an inconsistency management framework in the automated production systems domain. By means of distinct case studies, the applicability and feasibility of the inconsistency management framework are validated in Chapter 6. The main findings and results that can be derived from this dissertation are discussed in Chapter 7. Finally, a conclusion of this dissertation as well as an outlook on future research are given in Chapter 8.

Chapter 2.

Field of Investigation

This chapter provides an overview of the terms and definitions used in this dissertation. First, the domain of automated production systems is introduced, which is in the focus of this dissertation (Section 2.1). The current trend to overcome the complexity of engineering and operation of automated production systems is to shift from a document-centric view on the overall system towards using models [Sch06]. Hence, second, background knowledge on the field of model-based (systems) engineering as well as the respective technologies and methods in this field are given (Section 2.2). Third, the terms *inconsistency* and *inconsistency management* are defined (Section 2.3). Fourth, as the management of inconsistencies requires a highly flexible system that allows for specifying, diagnosing and handling inconsistencies, the notion of *knowledge-based systems* is introduced (Section 2.4). A conclusion of the findings provided in this chapter is given in Section 2.5.

2.1. Automated Production Systems

The following sections provide an overview of the domain of automated production systems. Therein, the term *automated production system* is elaborated first, followed by a discussion of the engineering and operation of automated production systems.

2.1.1. Definitions

Industrial production processes are and will be performed more and more by automated production systems, especially as the level of automation in such systems increases steadily [Str^{+}09]. Therein, *automation* aims at equipping technical systems to perform technical processes entirely or partly without the help of human work. Consequently, the *degree of automation*, according to IEC 60050-351 refers to the "proportion of automatic functions to the entire set of functions" [IEC13a, ref. 351-42-31] within the system and indicates, to which degree a technical process is realized in an automatic manner. Automation can therein reflect both *process automation* and *product automation*. Whereas process automation reflects processes that are performed on complex technical plants (e.g., power plants, manufacturing systems), product automation refers to processes that are performed on technical products (e.g., vehicles, consumer products) [LG99]. Although not limited to, this dissertation focusses on *process automation*.

The term *automation* consequently describes an interdisciplinary subject, which mainly focuses on the automation of *systems*. A *system*, according to IEC 60050-351, refers to a "set of interrelated elements [...] considered in a defined context as a whole and separated from their environment" [IEC13a, ref. 351-42-08]. A *technical process* performed on a technical system is specified as the "complete set of operations in a plant" [IEC13a, ref. 351-42-34] and, therefore, refers to all actions in a system that transform, transport or store material, energy or information [IEC13a, ref. 351-42-33]. Such technical processes can according to ISA 95 further be classified into discrete, continuous and batch processes [ISA10] as well as respective hybrid processes. Hence, a *technical system* describes the equipment that is needed to perform the respective technical process [LG99]. The relations between the technical process and the technical system are illustrated in Figure 2.1.

Definition 1 (Technical system) — *A technical system refers to the equipment that is needed to perform a technical process. Therein, a technical system contains the "set of interrelated elements [...] considered in a defined context as a whole and separated from their environment" [IEC13a, ref. 351-42-08].*

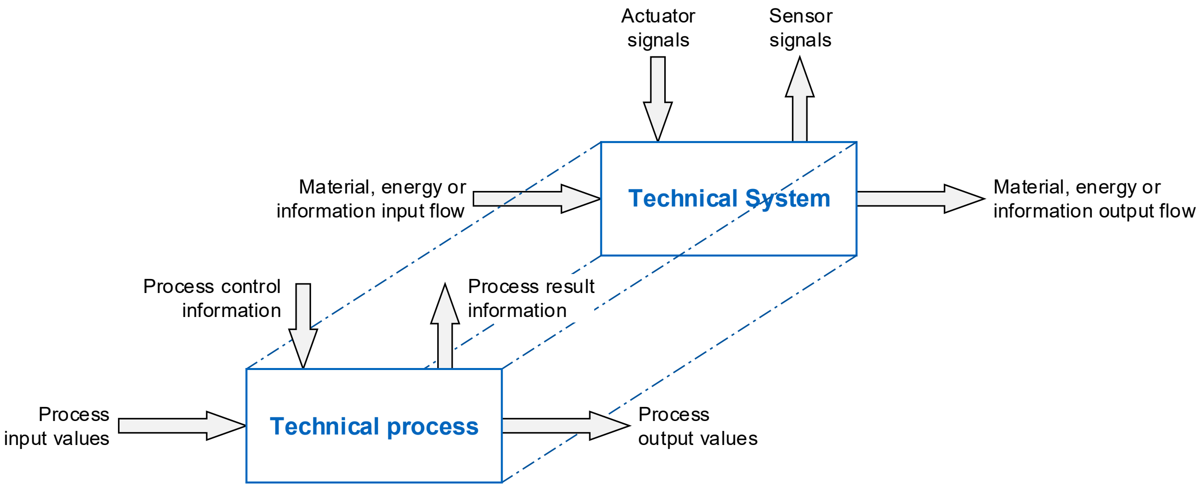

Actuator signals
Sensor signals
Material, energy or information input flow
Technical System
Material, energy or information output flow
Process control information
Process result information
Process input values
Technical process
Process output values

Figure 2.1. Relation between technical process and technical system [based on LG99, p. 4]

As can be seen from Figure 2.2, both technical system and technical process are essential parts of an automated production system. *Automated production systems* consist of "mechanical parts, electrical and electronic parts (automation hardware) and software, all closely interwoven, and thus represent a special class of mechatronic systems" [Vog$^+$15]. Hence, automated production systems reflect, according to Lauber and Göhner [LG99], the composition of:

- a *human-process communication* part, which serves as the interface between the user (e.g., end user, operator) and the automated production system,
- an *information processing and communication system* part, which reflects the processing hardware and software,
- the *sensors and actuators* contained in the technical system as well as
- the *technical process*, which is performed in the technical system.

Therein, the measured sensor values describe the current state of an automated production system, which is, in turn, controlled by the respective actuators. Consequently, knowledge on the technical system and the technical process is essential for both engineering and operation of such automated production systems.

Definition 2 (Automated production system) — *Systems in the automated production systems domain consist of "mechanical parts, electrical and electronic parts (automation hardware) and software, all [parts] closely interwoven, and thus represent a special class of mechatronic systems" [Vog$^+$15], .*

Furthermore, as can be seen from the structure of technical processes and systems as well as of automated production systems, the engineering and operation of automated production systems is a highly interdisciplinary process. Besides "classical" engineering activities that are common to any mechatronic system (e.g., mechanical, electrical and software engineering) as indicated in VDI 2206 [VDI04], further activities such as marketing, commissioning and maintenance need to be considered for automated production systems [Vog$^+$15].

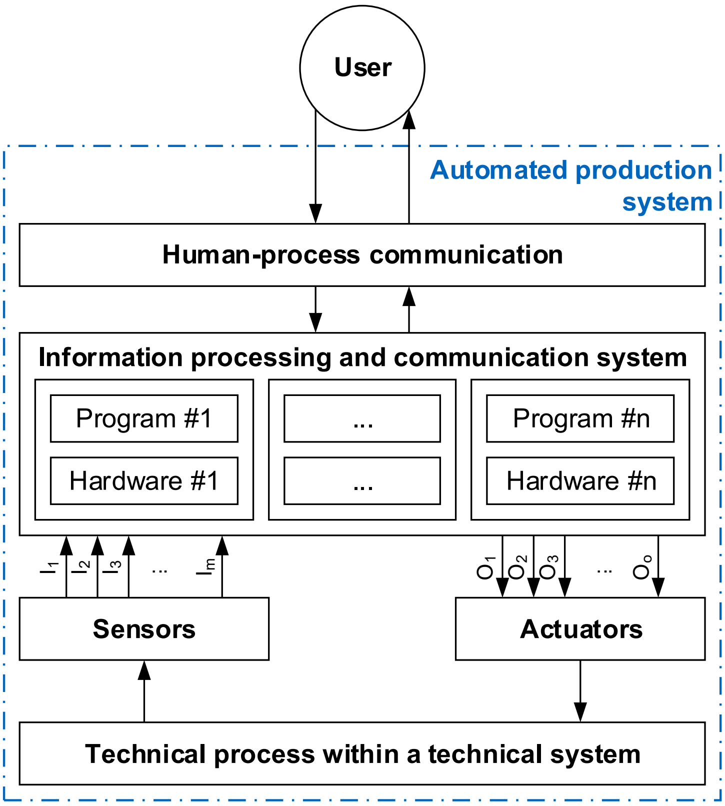

User
Automated production system
Human-process communication
Information processing and communication system
Program #1
Hardware #1
...
...
Program #n
Hardware #n
I1
I2
I3
...
Im
O1
O2
O3
...
Oo
Sensors
Actuators
Technical process within a technical system

Figure 2.2. Basic structure of an automated production system [based on LG99, p. 53]

2.1.2. Engineering and Operation of Automated Production Systems

Engineering in the automated production systems domain is a concurrent process and involves the different participating disciplines, e.g., mechanical, electrical and software engineering [FFV12]. One common engineering process model that is mostly applied in the mechatronic systems domain [VDI04] is the so-called *V model*, which is illustrated in Figure 2.3. Therein, *requirements* on the system form the starting point for the engineering process and, at the same time, serve as measures against which the mechatronic *product* is to be assessed. A cross-domain solution concept is developed during *system design*, followed by a *discipline-specific design*, in which discipline-specific detailed solutions are developed. Throughout the *system integration*, these discipline-specific solutions are integrated into a holistic discipline-spanning design. During the *assurance of properties* stage, the design process is continuously checked against the requirements defined beforehand – i.e., the fulfilment of the requirements is verified. The *modelling and model analysis* phase accompanies the entire engineering process, thereby describing the actions to create, manipulate and evaluate the documents and models being used by the discipline engineers.

In order to increase engineering efficiency and decrease the system's complexity, it is inevitable to re-use system components as much as possible [Vog$^+$15]. Therefore, to reduce engineering time, engineers aim at separating the engineering process into *project-independent* and *project-related engineering activities* [MJ12; VDI10]. Project-independent engineering activities focus on increasing the reuse of available parts of the engineering solution to shorten project durations and reduce engineering costs. Accordingly, project-related engineering activities aim at identifying the appropriate composition of these parts to fulfil the customer's needs. This separation is illustrated in Figure 2.4. During project-independent engineering activities, following an analysis of the solution elements that are required for possible systems, the requirements on the solution element are specified in a first

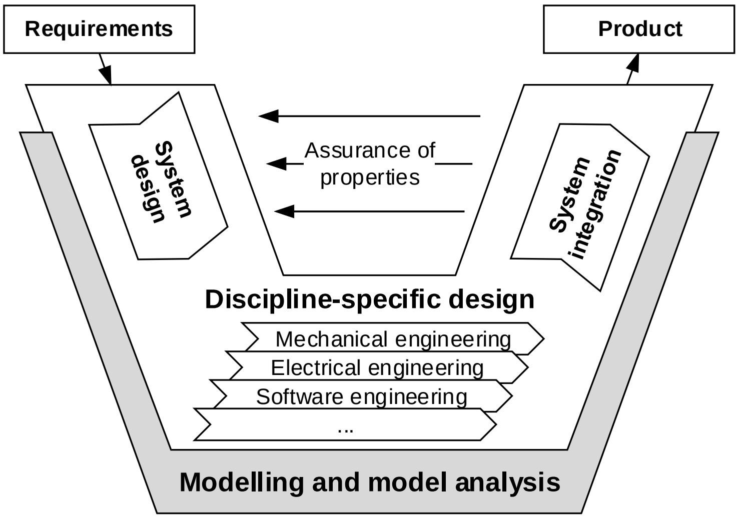


Figure 2.3. Overview of the V model [based on VDI04, p. 29]

step. Subsequently, the solution element is developed, tested/ approved and finally registered in a solution element repository. Using the solutions within the repository, customer-specific projects can take into account the solution elements during the phases of system and detail design (dashed arrows in Figure 2.4).

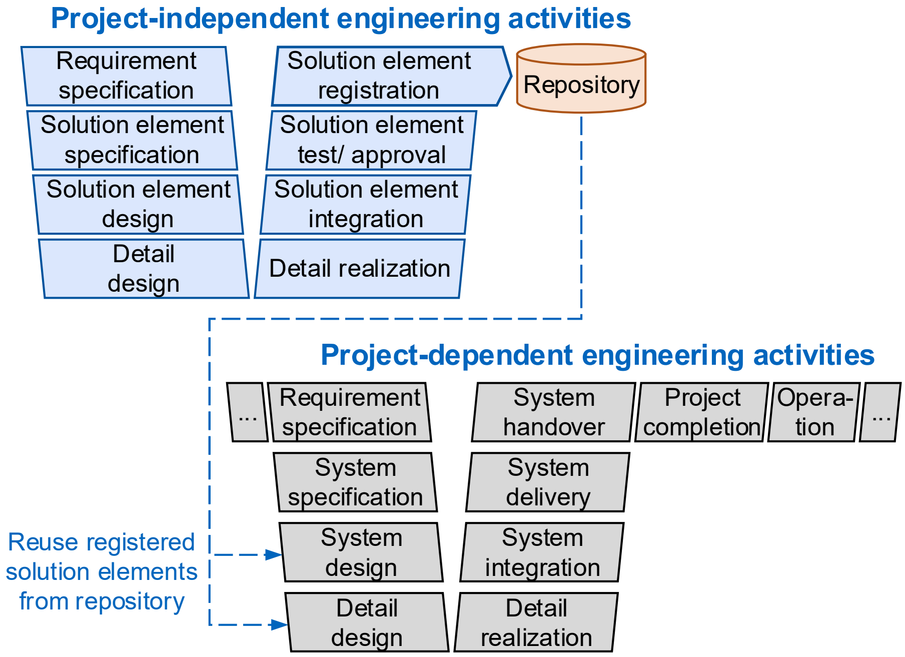

Project-independent engineering activities
Requirement specification
Solution element registration
Repository
Solution element specification
Solution element test/ approval
Solution element design
Solution element integration
Detail design
Detail realization
Project-dependent engineering activities
...
Requirement specification
System handover
Project completion
Opera-tion
...
System specification
System delivery
Reuse registered solution elements from repository
System design
System integration
Detail design
Detail realization

Figure 2.4. Separation into project-independent and project-related engineering activities [based on VDI10; Vog+15]

Whereas during the system design and integration phases, company- and/or project-specific formalisms and software tools are used to specify and integrate the system-level solution, discipline engineers mostly apply heterogeneous formalisms, abstraction mechanisms and software tools for the discipline-specific design of the automated production system. For instance, *mechanical engineering* incorporates a part-oriented design of the physical system and, hence, structural and/or geometrical data of the entire system are in focus of component lists, CAD drawings, etc. Contrary, the *electrical engineering* discipline aims at selecting the respective information processing and communication system of the automated production system, as well as the necessary sensors and actuators that are required to realize the technical process. Therefore, electrical circuit diagrams, terminal connection tables as well as electrical part lists are used to define the electrical subsystem. In the automated production systems domain, mostly Programmable Logic Controllers (PLCs) that are compliant to the IEC 61131 Standard [IEC03a] and that are characterized through a cyclic data processing behaviour, are and, according to the ARC Advisory Group [ARC15], will be state of the industrial practice for the next decades.

Finally, the objective of *software engineering* in the automated production systems domain is the implementation of the logical parts of the system, thereby defining the sequence of operations to be implemented for the technical process. For the software engineering discipline, the IEC 61131-3 Standard [IEC03b] is mostly state of the art in industry [TF11]. These signal-oriented programming languages impose – due to their monolithic structure – several restrictions regarding maintainability and, despite efforts towards including object-oriented programming aspects within IEC 61131-3 [IEC13b], the standard in its current version has not yet been fully established in industry. As a consequence, most of the machines and plants that are currently in operation are implemented (and, hence, maintained) in its older version or in vendor-specific implementations of the standard. Furthermore, although novel approaches such as multi-agent systems are being developed for IEC 61131-3 in research, they are not yet broadly accepted for software development in industry [Lei13]. IEC 61499 [Vya11; IEC12] provides improved maintainability as it increases the ability for software modularization and reuse by means of a component-based approach. However, “IEC 61499 has [still] a long way in order to be seriously considered by the industry” [Thr12].

The IEC 61131-3 [IEC13b] defines five programming languages – therein containing three graphical languages as well as two textual languages. The graphical programming languages are Function Block Diagram (FBD), which describes networks of connected logical or arithmetic expressions, Ladder Diagram (LD), which is similar to electrical circuit diagrams, and Sequential Function Chart (SFC), which allows the graphical description of a sequence of steps that are connected via directed transitions. The textual languages of IEC 61131-3 comprise Structured Text (ST), a language similar to the high-level programming language PASCAL, and Instruction List (IL), a low-level assembler-like programming language. The use of these languages is strongly dependent on a multitude of factors, e.g., customers’ requirements, programmers’ background, application context (FBD, LD and IL are appropriate for bit logic and often used for implementing, e.g., device drivers; ST is especially used for complex arithmetic, array or string operations and SFC is mainly applied for sequential operations and to structure the first level of a process to be implemented) as well as acceptance in destination country. Besides the IEC 61131-3 standard itself, huge effort is done within the Technical Committees of PLCopen, a vendor- and product-independent worldwide association, towards defining standardized exchange formats within PLCopen XML [PLC09b], towards certification of IEC 61131-3 environments [PLC09a] as well as, most recently, towards identifying common coding conventions for IEC 61131-3 [PLC16].

2.2. Model-Based (Systems) Engineering

Within the subsequent sections, the terms related to *model-based (systems) engineering* are elaborated deeply. First, common definitions in this field are shown. Second, both the software perspective (model-based engineering) and the systems perspective (model-based systems engineering) are discussed for the automated production systems domain.

2.2.1. Definitions

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, easing the software and system development, became a key issue to develop and handle complex systems. A model therein is often referred to as an abstraction of reality, which focuses on a certain point of interest.

According to the Model-Driven Architecture (MDA) introduced by the Object Management Group (OMG), a *model* is "[...] information selectively representing some aspect of a system based on a specific set of concerns" [Obj14]. Respectively, two essential parts can be found in this definition: First, information in models is *selective*: models reduce the reality to a subset of information that is believed to be interesting for the respective scope. Second, a specific set of *concerns* is the foundation for creating a model – consequently, models are created according to a specific objective. According to the MDA [Obj14], such models are subject to a specific system under investigation, respective rules that must be followed by that system as well as the meaning of the terms used for the model. Especially in the domain of automated production systems, "[m]odels are required for the conceptual design, implementation, testing, optimization and diagnosis" [Vog^{+}14a].

Definition 3 (Model) — *A model refers to an abstraction of reality, which focuses on a certain point of interest. Models are, hence, created according to a specific objective and subject to a specific system under investigation [Obj14].*

Accordingly, to be useful for the respective stakeholders, any model needs to be represented in "a way that communicates information about a system among involved stakeholders [...]" [Obj14] – hence, allowing for correct interpretation of what has been modelled. In order to achieve such a common interpretation, the terms and rules to be followed by the models must be captured within a *modelling language*. Commonly known modelling languages are, e.g., the Unified Modeling Language (UML) for software development, the Systems Modeling Language (SysML) for systems engineering as well as the Business Process Model and Notation (BPMN) for business process modelling. Therein, both the *syntax* and *semantics* of a model constitute the modelling language. *Syntax* refers to the structure of the modelling languages, describing the elements to be used within that language. Therein, it can be distinguished between the *abstract syntax*, describing the essential parts of the language, and the *concrete syntax*, defining what these parts look like. In contrast, *semantics* is about the meaning of the modelling language, i.e., the definition of what the model elements mean in the context of the model.

Definition 4 (Modelling language) — *A modelling language refers to the terms and rules to be followed by a model. Essential parts of such a modelling language are its* syntax *and* semantics*: Whereas the* syntax *refers to the structure of the modelling language (i.e., defining the elements to be used in that language (*abstract syntax*) as well as their (visual) representation (*concrete syntax*)), its* semantics *describe their context-specific meaning.*

Consequently, essential to formulate such modelling languages is the *abstract syntax*, which defines the elements to be used within the modelling language, as well as the *rules* to combine these elements to a valid model. In the following, the term *metamodel* is used for these two parts.

Definition 5 (Metamodel) — *A metamodel refers to the model of a model, which represents the formal definition of a modelling language. Essential to the metamodel is the* abstract syntax *of the modelling language, describing the structural elements as well as the rules to combine these elements to valid models.*

The most popular and commonly accepted standard to define such metamodels is the Meta Object Facility (MOF) [OMG15a] defined by the OMG. Therein, the MOF is part of the MDA and provides the means to define the abstract syntax of a modelling language through a simplification of UML's class modelling capabilities. Through MOF, interchangeable models can be created using the Extensible Markup Language (XML) Metadata Interchange (XMI) [Obj15] standard, and additional capabilities are provided such as model-to-model transformations [Obj16] or model-to-text transformations [Obj08]. The layered architecture of MOF is illustrated at the example of UML 2.5 [OMG15c] in Figure 2.5. Therein, the four meta layers of UML 2.5 are illustrated: Whereas the *M0* layer refers to the actual object (e.g., runtime instances), the *M1* layer refers to the model of the reality – in the example, to the UML model. The *M2* layer above the model layer represents the metamodel layer, in which the modelling language's metamodel is defined (i.e., the UML 2.5 metamodel). Finally, the *M3* layer refers to the metametamodel – which is, in the case of UML, MOF. However, although it is often referred to a "four-layered architecture", MOF is not restricted to a certain maximum number of layers – according to the standard [OMG15a], any number of layers that is greater than or equal to two is supported.

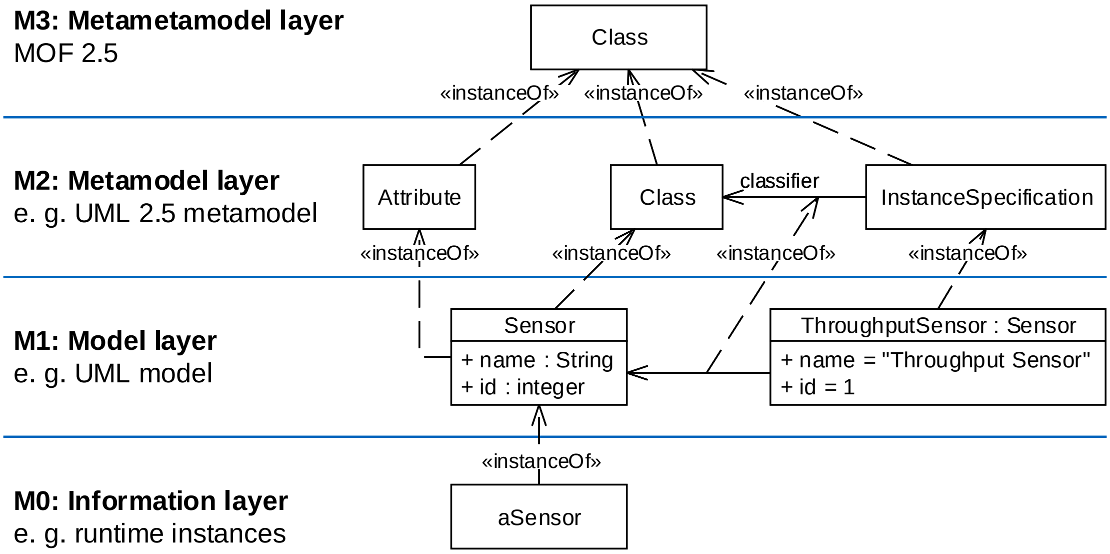


Figure 2.5. Illustration of the Meta Object Facility (MOF) [based on OMG15a] meta-layer architecture at the example of UML 2.5 [OMG15c]

Especially due to the manifold support in the field of modelling, models rapidly became an appropriate way in developing and maintaining (software) systems. Consequently, from a software perspective, Model-Based Engineering (MBE) serves as the basis to develop integrated models for the purpose of, e.g., analysis, code generation, etc. According to the International Council on Systems Engineering (INCOSE), *Model-Based Systems Engineering (MBSE)* is "[...] the formalized application of modelling to support system requirements, design, analysis, verification and validation activities [...]" [INC07]. Hence, instead of using documents (e.g., mechanical Computer-Aided Design (CAD) drawings, electrical circuit diagrams, etc.), a model-centric approach is envisioned, in which integrated models are used to establish an interdisciplinary engineering process. As a consequence, MBSE is seen as a life cycle-spanning process, which starts in early requirements analysis phases and allows for a fully integrated and consistent application of models until late engineering phases.

Definition 6 (Model-Based Systems Engineering) — *MBSE refers to "[...] the formalized application of modelling to support system requirements, design, analysis, verification and validation activities [...]" [INC07].*

2.2.2. Model-Based (Systems) Engineering in the Automated Production Systems Domain

This section discusses the different MBSE approaches that are currently emerging for the engineering of automated production systems[1].

Motivated by the example of classical computer science, MBE approaches were adopted rapidly for software development in the automated production systems domain. Therein, the UML is one of the most wide-spread languages used for software development. In its current version, UML 2.5 [OMG15c] specifies both *structure* and *behavior* diagrams to aid in developing software. Adoptions in the automated production systems domain are, among others, available for software generation [WV11] as well as for automated test case generation [RV17] – especially due to the fact that UML can increase comprehensibility and flexibility for software development [Vog15].

In order to support the interdisciplinary development of systems, hence incorporating further disciplines such as the electrical and mechanical engineering disciplines, SysML is increasingly applied for the automated production systems domain [Bas^{+}11; KV13; Thr13]. Additionally to the diagrams available in UML, SysML allows to specify the interdisciplinary system with its overall and internal structure. Moreover, diagrams to model requirements as well as parametric dependencies are supported, thereby putting more emphasis on the entire *system*. Adoptions to the automated production systems domain have been made, e.g., to support overall systems engineering while integrating simulation aspects [BFB14], for an integrated view on the entire mechatronic system [KV13] as well as for developing software agent knowledge bases [Sch^{+}13b].

2.3. Inconsistencies and Inconsistency Management

Whereas the previous sections focuses on the introduction of the terms related to the automated production systems domain as well as to Model-Based (Systems) Engineering, the arising need to address potentially occurring *inconsistencies* and, hence, the *management of inconsistencies* are discussed in the following. First, the terms *inconsistency* and *inconsistency management* are discussed to provide a basic understanding of this field. Second, different dimensions and types of inconsistencies that can occur in the automated production systems domain are elaborated to set the framework of this dissertation.

2.3.1. Definitions

In related research works, 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 [SZ01], therefore leading to a situation “in which a set of descriptions does not obey some relationship that should hold between them” [NER00]. If no known contradictions can be identified, the model (or the set of models) is identified to be *consistent*. Although a multitude of different definitions of inconsistencies exists, all these definitions “share a common property: that a state of *conflict*, marked by the presence of a *contradiction*, *illogical* statement or *disharmony* exists” [Her15]. One may argue that the occurrence of an inconsistency necessarily leads to an error during engineering – however, “[i]nconsistencies may have both positive and negative effects on the system development life-cycle” [SZ01]. According to Spanoudakis and Zisman [SZ01], on the negative side of inconsistencies are increased engineering time and cost as well as difficulties in maintaining the system, whereas positive effects are, e.g., to indicate aspects that need further elaboration or to simplify the evaluation of system alternatives.

Definition 7 (Inconsistency) — *An inconsistency refers to “a state of* conflict*, marked by the presence of a* contradiction, illogical *statement or* disharmony*” [Her15]. Hence, inconsistencies*

[1] A detailed discussion of different MBSE approaches can be found in Chapter 4.

lead to a situation "in which a set of descriptions does not obey some relationship that should hold between them" [NER00].

Inconsistencies are mostly the result of collaborative work between different stakeholders from different domains. Consequently, in Mens et al. [MSD06], the following four reasons for inconsistencies being introduced during model-based (systems) engineering are identified: (1) parallel development of disparate models by different persons, (2) poor understanding of interdependencies between models, (3) unclear or ambiguous requirements at an early stage during engineering, and (4) incomplete models due to still unknown, but essential information. As a consequence, the need to cope with inconsistencies arises, if models are created and maintained independently within collaborative projects [HEZ10]. Especially as different stakeholders from different disciplines are involved during engineering, they use different formalisms, models and tools in order to express their view on the system under consideration – and hence, form "separate, but interdependent models" [Gau$^+$09]. Therefore, the resulting models *overlap*: they "incorporate elements which refer to common aspects of the system under development" [SZ01]. Consequently, all these reasons of inconsistencies are mainly related to *overlapping* information; such *overlaps* are especially important for inconsistency management.

Definition 8 (Overlap) — *Overlaps are defined as statements that are made in disparate models, but, according to Spanoudakis and Zisman [SZ01] "incorporate elements which refer to common aspects of the system under development". Consequently, such overlaps are especially important for inconsistency management.*

Within Finkelstein et al. [Fin$^+$94], it is argued that the occurrence of inconsistencies is inevitable, yet acceptable for engineering. As a consequence, the need to *manage* such inconsistencies is identified [Fin$^+$94]. In Nuseibeh et al. [NER00], an inconsistency management framework is introduced (cf. Figure 2.6), which involves six distinct steps: After *monitoring for inconsistencies*, a detected inconsistency is *diagnosed*. The *diagnosis* of inconsistencies therein involves their *location* (What elements are inconsistent?), *identification* (What is the cause for an inconsistency?) and *classification* (What kind of inconsistency occurred? What is its expected impact?). In order to diagnose inconsistencies, expected impacts need to be *measured*. After an inconsistency is diagnosed, the *handling* step begins. Therein, according to Nuseibeh et al. [NER00], either *ignoring*, *tolerating* or *resolving* are appropriate strategies to handle an inconsistency. The respective handling actions need to be *analysed* regarding their impact and risk. If a stakeholder (i.e., the person responsible for managing inconsistencies) selects a handling action, the *consequences* of this action need to be *monitored*. Consequently, four important parts can be derived: (1) the incorporation of *different types of inconsistencies*, (2) the *definition and identification of overlaps* between models, (3) the *diagnosis of inconsistencies*, and (4) the *handling of diagnosed inconsistencies*. Hence, these steps are considered as essential for inconsistency management throughout this dissertation.

Definition 9 (Inconsistency management) — *Based on the results of Nuseibeh et al. as well as Spanoudakis and Zisman [NER00; SZ01], it can be concluded that inconsistency management incorporates four essential components: (1) the incorporation of* different types of inconsistencies*, (2) the* definition and identification of overlaps *between models, (3) the* diagnosis of inconsistencies*, and (4) the* handling of diagnosed inconsistencies.

In order to manage inconsistencies, different strategies can be employed. Among others[2], one commonly employed strategy is rule-based inconsistency management. Therein, consistency rules refer to constraints that need to be fulfilled by the models under investigation to consider them as consistent – hence, the models are said to be inconsistent if the consistency rule is not satisfied by

[2] A detailed discussion of different inconsistency management strategies can be found in Chapter 4.

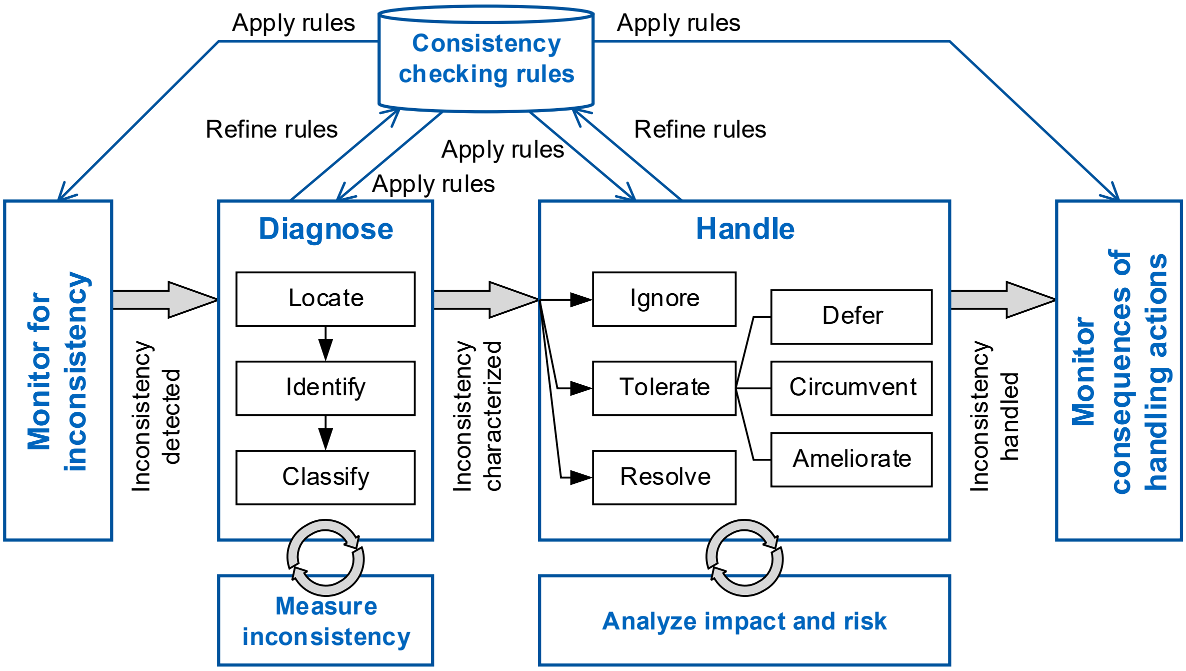


Figure 2.6. Overview of the building blocks of an inconsistency management framework [adapted from NER00]

the models [SZ01]. Accordingly, "[c]entral to [...] [an inconsistency management] framework is the explicit use of a set of consistency rules" [NER00].

Definition 10 (Consistency rule) — *A consistency rule refers to constraints that need to be fulfilled by the models under investigation – hence, the models are said to be inconsistent if the consistency rule is not satisfied by the models [SZ01]. Accordingly, "[c]entral to [...] [an inconsistency management] framework is the explicit use of a set of consistency rules" [NER00].*

However, one special property of technical systems is that it is impossible to ensure full consistency – "the main reason is that we lack perfect knowledge about the processes and the phenomena in nature" [Her+11]. Consequently, only a small portion of inconsistencies can be managed for technical systems – that is, only known inconsistencies can be identified. One can think of these known inconsistencies as *patterns*, which define the conditions for the existence of an inconsistency [Her+11]. Using such inconsistency patterns, models can be queried and, by means of pattern matching, inconsistencies can be diagnosed. Once an inconsistency has been identified, the inconsistent part of the models can be handled by ignoring, tolerating or resolving the inconsistent part.

Definition 11 (Inconsistency pattern) — *An inconsistency pattern describes the conditions for the existence of an inconsistency [Her+11]. Using such inconsistency patterns, models can be queried for defined inconsistency patterns and, by means of pattern matching, inconsistencies can be diagnosed. Once an inconsistency has been identified, the inconsistent part of the models can be handled by ignoring, tolerating or resolving the inconsistent pattern.*

2.3.2. Dimensions of consistency

In the related literature, different dimensions of consistency can be found, which are discussed in the following.

Intra-model and inter-model consistency. In Huzar et al. [Huz+05], *intra-model* and *inter-model consistency* are identified as important aspects in UML models. For instance, *intra-model* inconsistencies result from many artefacts that describe different aspects of a system, but are inconsistent, e.g., due to "the imprecise semantics of the UML" [Huz+05]. Such intra-model inconsistencies are especially expected between structural and behavioural aspects of the system. In contrast, *inter-model* inconsistencies arise between different models, e.g., in case of refinements between models. Whereas intra-model inconsistencies are often managed by existing software tools (e.g., Eclipse Papyrus [Ecl15] and NoMagic MagicDraw [NoM16] in case of UML and SysML), inter-model inconsistencies are especially challenging due to the heterogeneous nature of models in the automated production systems domain (cf. Section 2.1).

Metamodel, model and instance consistency. Following the layered architecture of OMG's MOF, consistency can also be identified in these different modelling layers as discussed in Section 2.2, namely within *metametamodels*, *metamodels*, *models* as well as *instances*. For the software engineering domain, inconsistencies can arise on the *model* level, between the *model* and *instance* level or at the *instance* level [Van+03]. Further potential inconsistencies may analogously arise between the *metamodel* and the *model* as well as on each respective layer of the MOF. However, whereas consistency is mostly ensured between layer L_i of the MOF and its child layer L_{i-1} by means of common software tools such as the Eclipse Modeling Framework (EMF), it is inevitable to incorporate inconsistencies between disparate but heterogeneous models (cf. inter-model consistency).

Intensional and extensional consistency. Based on the notion of *metamodel, model and instance consistency*, it can be concluded that, from the specification of a *metamodel* (respectively *model* and *instance*), the constraints that need to be fulfilled to ensure an inherent (i.e., *intensional*) consistency can be derived. In the related literature on inconsistency management, this *intensional consistency* is often refereed to as well-formedness constraints that define the correspondences between types of model elements [RJV09]. However, besides this intensional consistency, further constraints can be defined that result, e.g., from domain- or company-specific assumptions. These constraints are referred to as *extensional consistency* constraints that are defined as correspondences between particular model elements [RJV09].

Structural and behavioural consistency. With the terms *structural* and *behavioural* consistency, either structural or behavioural aspects of software models are described [Van+03]. Structural or behavioural inconsistencies, hence, arise "when the structure of the system is incomplete, incompatible or inconsistent with respect to existing behaviour" [Van+03]. For instance, structural inconsistencies refer to missing instance definitions within software models, whereas behavioural inconsistencies arise if incompatible behaviour definitions are observed.

Horizontal and vertical consistency. In Van Der Straeten et al. [Van+03], the notion of *horizontal*, *vertical* and *evolution* consistency is introduced. Therein, similar to inter-model and intra-model consistency, *horizontal consistency* refers to "consistency between different models within the same version" [Van+03]. *Vertical consistency* is used to describe the relations between one model and its refinements, which describes more details of this model. *Evolution* consistency, analogously, refers to the consistency between different versions of the model.

External and internal consistency. In Herzig et al. [Her+11], it is concluded that *internal* and *external consistency* exist. Whereas *internal consistency* relates to axiomatic systems (e.g., mathematical consistency, consistency to logic systems), *external consistency* requires the models to be consistent to reality. It is moreover concluded that, whereas internal consistency is well understood and, hence, can be managed, external consistency often refers to poorly understood (e.g., physical) phenomena, which can hardly be managed by an inconsistency management framework [Her^{+}11].

2.3.3. Classification of inconsistencies

Orthogonal to the dimensions of inconsistencies, different types of inconsistencies can be distinguished. For instance, *well-formedness*, *description identity*, *application domain*, *development compatibility* and *development process compliance* rules are, according to Spanoudakis and Zisman [SZ01], important for the software engineering domain. Whereas *well-formedness* rules must be fulfilled by the models to be legitimate models of the respective modelling language, *description identity* rules demand that totally overlapping elements must have identical descriptions. *Application domain* rules refer to relations that must hold between individuals in the specific application domain. *Development compatibility* rules refer to “rules which require that it must be possible to construct at least one model that develops further two or more other models or model elements and conforms to the restrictions which apply to both of them” [SZ01]; hence, these rules do not demand total but partial overlap between respective model entities. Finally, *development process compliance* rules describe practices, guidelines or standards that need to be followed by the models.

Similarly, in Nuseibeh et al. [NER00], *notation definitions*, *development methods*, *development process models*, *local contingencies* and *application domains* are defined as essential sources for consistency rules. Whereas *notation definitions* are sources for *well-formedness* rules, and *development methods* and *development process models* are essential to construct *development compatibility*, *description identity* and *development process compliance* rules, *application rules*, finally, lead to *application domain* rules. Interestingly, *local contingencies* refer to a rather weak descriptions of inconsistencies, which are not necessarily predetermined by the relationship, but by background knowledge such as synonym definitions.

Based on these different types of inconsistency rules, it can be derived that four essential types of inconsistencies can be identified for the automated production systems: (1) *notational rules* refer to syntactic integrity constraints of notations, (2) *conventional rules* refer to common practices that must be followed during engineering and (3) *correspondence rules* define, how entities in disparate models must relate to each other. Finally, (4) *domain-specific rules* refer to inconsistency rules that must not be violated in a specific application domain, e.g., the automated production systems domain. These types of inconsistencies are focused on in this dissertation.

2.4. Knowledge-Based Systems

As discussed in the previous section, especially in the automated production systems domain, the different types of inconsistencies that must be incorporated for inconsistency management can be manifold. Consequently, it is inevitable to provide appropriate means for flexibly specifying, diagnosing and handling inconsistencies. One means to provide such a flexible mechanism is the usage of *knowledge-based systems*, which are introduced in the following.

2.4.1. Definitions

An exclusively procedural software system requires to explicitly include the knowledge about the structure and semantics of a variety of models as well as inconsistencies in the software code. However, maintaining and evolving such a software system – for instance, for the purpose of incorporating additional types of inconsistencies, or additional types of models – is costly due to the sheer complexity. Any practical implementation of an inconsistency management framework for heterogeneous models, therefore, requires a high degree of flexibility and extensibility. Hence, a *knowledge-based system* is envisioned to be used for inconsistency management in this dissertation.

According to Akerkar and Sajja [AS10, pp. 18-19], a knowledge-based system "is a computer-based system that uses and generates knowledge from data, information and knowledge". By that, such a system is able to *understand* the information to be processed (e.g., by means of diagnosing inconsistencies) and to *decide* upon which action to be taken (e.g., by means of generating possible handling actions).

DEFINITION 12 (KNOWLEDGE-BASED SYSTEM) — *A knowledge-based system, according to Akerkar and Sajja [AS10, pp. 18-19], "is a computer-based system that uses and generates knowledge from data, information and knowledge".*

Knowledge-based systems consist of five essential parts (cf. Figure 2.7): The *knowledge base* together with the *inference engine* form the main component of a knowledge-based system and serve as the knowledge repository. Knowledge-based systems can either be updated automatically or manually by means of the *self-learning* component. The *explanation and reasoning* part provides information on what conclusions are drawn from the knowledge base. A *user interface* is essential to provide the user with an appropriate interface to the knowledge-based system.

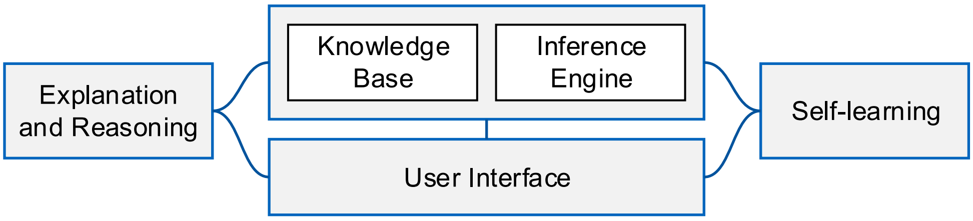


Figure 2.7. Essential parts of a knowledge-based system [adapted from AS10, p. 20]

As can be seen from Figure 2.7, the most important parts for a knowledge-based system are the *knowledge base* itself as well as the *inference engine*. The *knowledge base* consists of both a procedural and a declarative component [AS10, p. 33]. Declarative knowledge, hence, serves as the descriptive representation of knowledge, which consists of factual statements and information (i.e., *facts* and *rules*). In the context of inconsistency management, the *facts* refer to the knowledge specified in the different models, whereas the *rules* represent the rules (respectively: patterns) for inconsistency diagnosis and handling. Procedural knowledge, in turn, consists of common-sense and heuristic knowledge. For inconsistency management, this procedural knowledge consists of, e.g., the knowledge on how and what inconsistency diagnosis or handling results are presented to the user. The *inference engine* refers to the *knowledge base*, manipulates the knowledge and makes decisions on actions to be taken [AS10, p. 36]. By that, new facts are drawn from the rules and known facts in the knowledge base.

2.4.2. Knowledge Representation

As argued beforehand, essential to a knowledge-based system is its knowledge base. Consequently, it is essential to find an appropriate knowledge representation formalism that is capable of capturing the necessary modelled knowledge in the knowledge base.

Resource Description Framework (RDF). One formal language used to describe structured information and, hence, to represent knowledge in the context of Semantic Web Technologies is the RDF standardized by the World Wide Web Consortium (W3C) in [W3C14b]. RDF includes a suite of recommendations, which allow for representing information in the web. Originally, the goal of RDF is to allow applications to "exchange data on the Web while preserving their original meaning" [HKR10]. Hence, the original intention of RDF is close to the challenge of heterogeneous models – to describe heterogeneous knowledge in a common representational formalism. Therein, RDF is similar to conceptual modelling approaches such as class diagrams in that it allows for statements to be made about entities, e.g., *cylinder is a module and consists of a valve and a switch*. Such statements about entities are formulated by means of subject – predicate – object triples (e.g., *cylinder – is a – module*, *cylinder – consists of – switch*), thereby forming a directed, labelled graph. Such an exemplary RDF graph is visualized in Figure 2.8.

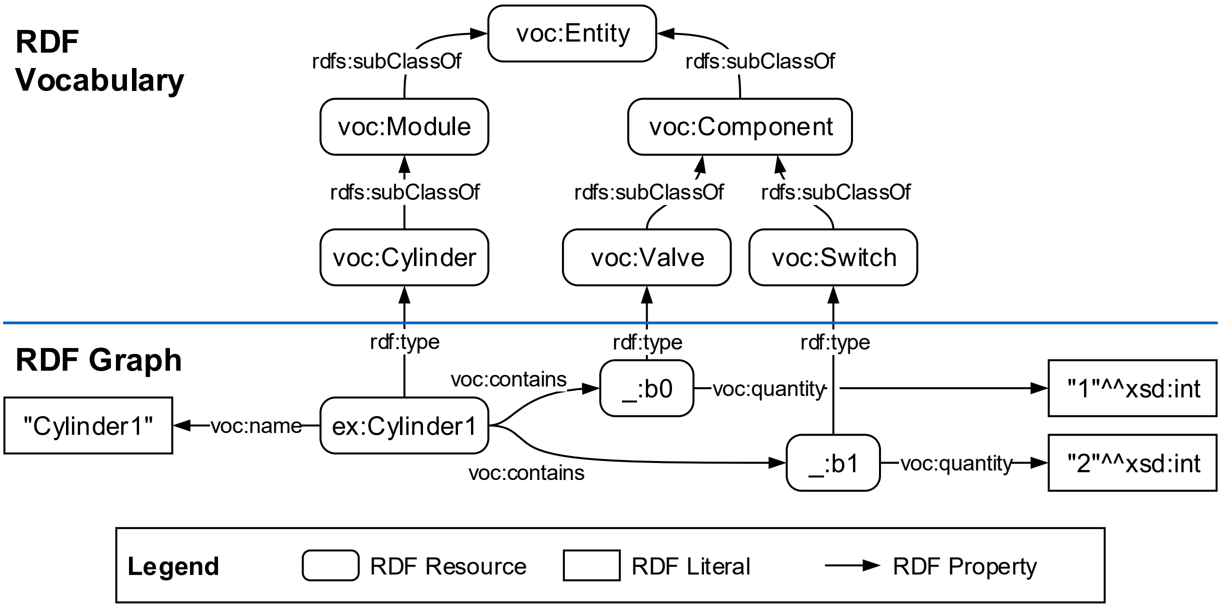


Figure 2.8. Exemplary RDF graph

To leave no room for ambiguities, RDF makes use of so-called *Uniform Resource Identifiers (URIs)* as unique names for entities (e.g., for the resources such as `voc:Module` and `ex:Cylinder1` as well as for the properties such as `voc:contains`). The different URIs being used in the RDF graph are moreover organized by means of *namespaces*, which predefine the URIs. For instance, in the example illustrated in Figure 2.8, the predefined vocabularies `ex` and `voc` contain the URIs to be used within the RDF graph.

The subject – predicate – object triples used within the RDF graph are, according, to the RDF specification [W3C14b], restricted as follows:

- *subjects* are either URI references or a blank node,
- *predicates* are URI references, and
- *objects* are either URI references, literals or blank nodes.

Therein, *literals* refer to either *plain literals*, which have a lexical form and an optional language tag (cf. literal `"Cylinder1"` in Figure 2.8), or to *typed literals*, which have a lexical form and a datatype URI (cf. literal `"1"^^xsd:int` in Figure 2.8). *Blank nodes* refer to RDF resources, for which no URI or literal is given, e.g., for `_:b0` and `_:b1` in Figure 2.8. These blank nodes are often used for, e.g., reification, complex attributes, etc. In the cylinder example, the blank nodes serve for describing the cardinality of contained components.

For the purpose of serializing RDF graphs, the W3C recommendation provides the common serialization formats *Turtle*, *N-Triples*, *N-Quads*, *JSON-LD*, *N3* and *RDF/XML*. For the purposes of this dissertation, a graphical representation of RDF graphs as illustrated in Figure 2.8 will be used together with the Turtle syntax [W3C14c]. The Turtle representation of the exemplary RDF graph as illustrated in Figure 2.8 is given in Listing 2.1.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix ex: <http://www.example.org/ns/example#>.
@prefix voc: <http://www.example.org/ns/vocabulary#>.

# RDF Vocabulary - Classes
voc:Entity    rdf:type        rdfs:Class .
voc:Module    rdfs:subClassOf voc:Entity .
voc:Component rdfs:subClassOf voc:Entity .
voc:Cylinder  rdfs:subClassOf voc:Module .
voc:Valve     rdfs:subClassOf voc:Component .
voc:Switch    rdfs:subClassOf voc:Component .

# RDF Vocabulary - Properties
voc:name      rdf:type        rdf:Property .
voc:contains  rdf:type        rdf:Property .
voc:quantity  rdf:type        rdf:Property .

# RDF Graph
ex:Cylinder1  rdf:type        voc:Cylinder ;
              voc:name        "Cylinder1" ;
              voc:contains [
                rdf:type      voc:Valve ;
                voc:quantity  "1"^^xsd:int  ] ;
              voc:contains [
                rdf:type      voc:Switch ;
                voc:quantity  "2"^^xsd:int  ] .
```

Listing 2.1 Exemplary RDF Turtle representation

RDF Schema (RDFS). As argued beforehand, RDF vocabularies represent collections of URIs with a clearly defined meaning. For such a meaning to be described in a machine-interpretable manner, besides specifying knowledge on instances (i.e., *assertional knowledge*), the RDF recommendation allows for specifying background information (i.e., *terminological knowledge*) by means of RDFS [W3C14d]. RDFS provides language constructs to formulate simple graphs containing class and property hierarchies as well as property restrictions. With its formal semantics, RDFS leaves no room for interpretation of what conclusions can be drawn from a given graph, thereby providing standard inference mechanisms for any RDFS-compliant graph. For instance, as in the cylinder example, a cylinder is specified to be a specialized class of module (i.e., `voc:Cylinder rdfs:subClassOf voc:Module`), each instance of cylinder will, hence, be classified to be a module instance as well. By that, simple inference is possible by means of RDFS.

RDFS can, hence, be used as a language to model simple ontologies, but provides limited expressive means and is not suitable to formulate more complex knowledge [HKR10]. Examples for knowledge that cannot be formulated in RDFS are phrases such as *Each Module consists of at least one component* and *Components are either actuators or sensors.*

Web Ontology Language (OWL). One possibility to formulate such more complex knowledge are rules that can be used to draw *conclusions* from a *premise* statement, i.e., by applying rules in the form of *IF premise THEN conclusion.* Another way to formulate complex knowledge is the use of the OWL [W3C12], which provides further language constructs defined with description logics based semantics. OWL moreover contains two sub-languages[3] to provide a choice between different degrees of expressivity, scalability and decidability – namely OWL Full and OWL DL. Therein, OWL DL enables maximum expressivity while maintaining decidability [HKR10]. The OWL DL formal (description logics based) semantics allow to define what conclusions can be drawn from an OWL DL ontology. However, for the purpose of managing inconsistencies in heterogeneous models of automated production systems, this dissertation will use the sufficient expressivity of RDFS.

2.4.3. Knowledge Processing

A set of specifications, which provide the means to retrieve and manipulate information represented in RDF, RDFS (or OWL, respectively) is the SPARQL Protocol and RDF Query Language (SPARQL). The primary components of the standard are the *SPARQL Query Language* [W3C13a] as well as the *SPARQL Update Language* [W3C13b]. SPARQL is in many regards similar to the well-known Structured Query Language (SQL), which is supported by most relational database systems.

SPARQL Query Language. The SPARQL Query Language provides the capabilities for querying required and optional graphs along with their conjunctions and disjunctions. A query consists of three major parts: *namespace* definitions being used within the query, a *clause* identifying the type of the query and a *pattern* to be matched against the RDF data. SPARQL is highly expressive and allows for the formulation of required and optional patterns, negative matches, basic inference (e.g., property paths to enable transitive relations), conjunctions and disjunctions of result sets as well as aggregates, i.e., expressions over groups of query results. Four disparate query types can be used in SPARQL:

- *SELECT queries* return values for variable identifiers, which are retrieved by matches to a particular pattern against the RDF graph,
- *ASK queries* return a Boolean variable that indicates whether or not some result matches the pattern,
- *CONSTRUCT queries* allow for substituting the query results by a predefined template for the RDF graph to be created, and
- *DESCRIBE queries* return a single RDF graph containing the relevant data about the result set. As the "relevance" of data is strongly depending on the specific application context, SPARQL does not provide normative specification of the output being generated by DESCRIBE queries.

An exemplary SELECT query is illustrated in Listing 2.2. Therein, each instance `?i` of the class `voc:Entity` is retrieved. Moreover, the optional pattern `?i voc:name ?iName` is queried against the graph – hence, the names of all entities are retrieved and (if applicable) bound to the variable `?iName`. Finally, the Boolean comparison `bound(?iName)` checks whether `?iName` is bound (i.e., whether a name is available for the entity) and binds the result to the variable `?hasName`. In the exemplary RDF graph, it can hence, be retrieved, whether all entities have an assigned name or not – the blank nodes `_:b0` and `_:b1` can be identified to not have a name.

[3] Note that, in addition to OWL DL and OWL Full, there are three profiles for a variety of applications – namely OWL EL, QL and RL – which, however, are out of the scope of this dissertation.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://www.example.org/ns/example#>
PREFIX voc: <http://www.example.org/ns/vocabulary#>

SELECT ?i ?iName ?hasName
WHERE {
  ?i rdf:type voc:Entity .
  OPTIONAL { ?i voc:name ?iName } .
  BIND ( bound(?iName) AS ?hasName ) .
}
```

Listing 2.2 Exemplary SPARQL SELECT query

SPARQL Update Language. The SPARQL Update Language makes use of a syntax derived from the SPARQL Query Language and allows for performing update operations on an RDF graph. Therein, it provides the means to update, create and remove RDF triples. Hence, five distinct update operations can be used in SPARQL:

- *INSERT DATA* operations allow for adding triples into the RDF graph,
- *DELETE DATA* operations provide the means to delete triples from the RDF graph,
- *DELETE/INSERT* operations allow for pattern-based manipulation of RDFgraphs,
- *LOAD* operations provide the means to load an RDF graph into a graph store, and
- *CLEAR* operations provide the means to remove all triples from a graph.

An exemplary INSERT operation is illustrated in Listing 2.3. Therein, based on the previously introduced SELECT query, for each instance `?i` that does not have an associated name, the name `"NewName"` is initialized. Hence, after executing this update operation, the blank nodes `_:b0` and `_:b1` receive an associated name.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://www.example.org/ns/example#>
PREFIX voc: <http://www.example.org/ns/vocabulary#>

INSERT { ?i voc:name "NewName" . }
WHERE {
  ?i rdf:type voc:Entity .
  OPTIONAL { ?i voc:name ?iName } .
  BIND ( bound(?iName) AS ?hasName ) .
  FILTER ( !?hasName ) .
}
```

Listing 2.3 Exemplary SPARQL INSERT operation

2.5. Summary

As discussed within this chapter, the automated production systems domain is a highly interdisciplinary domain. Although model-based (systems) engineering techniques are more and more used in this domain, especially due to the manifold modelling languages, abstraction levels as well as software tools, inconsistencies (that is: conflicts between the models) are likely to occur in these overlapping models, making an appropriate approach for inconsistency management necessary. However, in order to provide such an inconsistency management, a high degree of flexibility and extensibility is necessary and, therefore, a knowledge-based system is envisioned to be used for inconsistency management in this dissertation. In order to provide appropriate technologies to realize such an inconsistency management, RDF/RDFS is used to represent the knowledge encoded within models, together with SPARQL to query and manipulate the knowledge.

Chapter 3.

Requirements and Simplifying Assumptions

Today's engineering of automated production systems involves a multitude of disciplines. As a consequence, collaborative effort of different stakeholders – among others, mechanical engineers, electrical engineers and software engineers – is essential for the success of an engineering project in this domain. As discussed in Chapter 2, this collaborative process increases the probability that inconsistencies are introduced in the interdisciplinary engineering models and documents. In contrast, as shown by industry surveys such as the one performed by Schmidt et al. [Sch+14], automated inconsistency management is up to now "[...] barely implemented in current [engineering] tools, but must be available in the next two years".

The overall aim of this dissertation is to develop a framework for (semi-)automated inconsistency management, thereby supporting the different discipline-specific experts in the domain of automated production systems appropriate in specifying, diagnosing and handling inconsistencies. Consequently, the inconsistency management process must be automated as much as possible to decrease the engineering effort. However, a completely automated inconsistency management process is neither possible nor desirable [Her+11]. Rather, it is aimed at benefiting from managing inconsistencies by (1) showing stakeholders the *hot spots* in their engineering models to early identify potential ambiguities and errors and (2) supporting stakeholders by indicating, which parts of the engineering models need further elaboration for the purpose of simplifying the evaluation of engineering alternatives.

While developing the inconsistency management framework, the proposal in [NER00] is followed, in which the incorporation of different types of inconsistencies, the definition and identification of overlaps between models, the diagnosis of inconsistencies and the handling of inconsistencies as essential parts of the inconsistency management process (see Definition 9 in previous Chapter 2) are described. However, as argued beforehand, it is not possible, neither desirable to ensure full consistency of engineering models [Her+11]. As a consequence, the best one can do is to specify known inconsistencies, and search for those in the respective engineering models. Consequently, a pattern-based inconsistency management approach is followed, which makes use of *inconsistency patterns* (see Definition 11 in Chapter 2), that are matched against the engineering models.

The requirements that need to be fulfilled by such a pattern-based inconsistency management approach (Section 3.1) as well as the simplifying assumptions imposed for the purpose of developing the approach (Section 3.2) are described in the following. These requirements are mostly based on the related literature and have been refined during discussions with several researchers in this field within previous research work [Fel+15b; Fel+15a; Fel+16c]. A summary of this chapter is given in Section 3.3.

3.1. Requirements

In the following, the requirements that must be fulfilled by an inconsistency management approach for the automated production systems domain are described. First, requirements are introduced that need to be fulfilled by a model knowledge base, which contains all the models under consideration for an engineering project (Section 3.1.1). Subsequently, the requirements regarding the diagnosis

(Section 3.1.2) and handling of inconsistencies (Section 3.1.3) are introduced. As the means to measure diagnosed inconsistencies and to assess the impact of handling actions is essential to support stakeholders in identifying the correct procedure, the requirements regarding measurement and assessment capabilities are introduced in Section 3.1.4. Finally, the requirements that result from the operationalization of an inconsistency management approach in the automated production systems domain are discussed in Section 3.1.5.

3.1.1. Requirements Regarding the Model Knowledge Base

During the engineering of automated production systems, a multitude of different stakeholders from different disciplines is involved [Vog$^+$15]. These stakeholders make use of different modelling formalisms, languages and tools [Gau$^+$09; Bro$^+$10]: e.g., whereas mechanical engineers make use of Computer-Aided Design (CAD) systems focusing on the physical structure of an automated production system, electrical engineers work with electrical circuit diagrams and software engineers implement the system's logic within IEC 61131-3. Moreover, as the amount of available information necessary to select or reject system alternatives changes during the life-cycle of an engineering project (see discussions in Section 2.1), these stakeholders not only make use of different notations, but also of different levels of abstraction to express their concerns [Jäg$^+$12]. For instance, during requirements engineering of automated production systems, less information is available than during systems design – consequently, the level of detail grows throughout the engineering process. The resulting heterogeneity of engineering models poses a major challenge [Gau$^+$09; Bro$^+$10], as the composition of the different models is difficult if not impossible. Some authors even argue that this heterogeneity "make[s] collaboration more difficult and more risky" [WB12]. As a consequence, a common *model knowledge base* must form the basis to allow for inconsistency management in the multitude of heterogeneous models in automated production systems engineering (see Requirement 1). Contrary to related research works, which focus on automated extraction of models from existing engineering documents [Arr$^+$16], in this dissertation, it is focused on the information content within the engineering documents, namely on the engineering models behind these documents.

Requirement 1 (Model knowledge base) — *An approach for inconsistency management in the automated production systems domain must provide a respective model knowledge base; that is, it must allow for processing across heterogeneous models.*

In order for such a model knowledge base to be applicable for inconsistency management, there are typically two alternatives that can be aimed at: First, synchronizations between models can be created in order to ensure that the models are free of inconsistencies. However, the creation and maintenance of these synchronisations requires huge efforts, as for n models, either $n \cdot (n-1)$ uni-directional synchronisations or $n \cdot (n-1)/2$ bi-directional synchronisations are necessary. Second, an integrated knowledge base can be used, which captures the modelled entities to be considered for inconsistency management. Consequently, it is essential to provide a common syntax (see Requirement 1.1) for the models under consideration. By means of such a common syntax, a common representational formalism is ensured, which (1) reduces the amount of necessary synchronisations to n bi-directional synchronisations between the model-specific formalisms and the common representational formalisms and (2) allows one to query the models for diagnosing inconsistencies as well as to manipulate the models for the purpose of handling inconsistencies in the common formalism.

Requirement 1.1 (Common syntax) — *A common syntax must be provided to allow for symbolically processing across heterogeneous models.*

If the involved models were available in a common representational formalism, respective mechanisms to manage inconsistencies in these models could be put in place. However, as discussed in Chapter 2, the respective models are subject to commonalities. For instance, a sensor introduced to the mechanical subsystem of an automated production system has its respective representation in the electrical subsystem as well as in the respective control software. Hence, the different models share *common concepts*. In case of inconsistency patterns applicable for each of the involved disciplines, one would have to specify the respective patterns for each discipline. For instance, if each representation of the sensor must follow a given naming convention, this naming convention would need to be specified for each of the involved disciplines. Accordingly, a *common semantics* of the involved models is necessary (see Requirement 1.2) – first, to capture the concepts that are relevant for the discipline and, second, to capture the discipline-spanning concepts of the models. By that, the sensor would have its representation as a *sensor* concept in the mechanical, electrical and software discipline as well as in a common mechatronic discipline.

Requirement 1.2 (Common semantics) — *A common semantics must be provided to (1) allow for capturing discipline-specific concepts of the involved models and (2) allow for capturing the discipline-spanning concepts of the involved models.*

By means of a common syntax and semantics of the models under consideration (see Requirements 1.1 and 1.2), the involved models can be put into a common representational formalism that allows the description of common concepts of the respective models. However, in order to capture the overlaps between the entities in the disparate models [SZ01], an explicit representation of the links between the model entities is necessary (see Requirement 1.3). By that, a user can specify that the sensor entity in the mechanical model is said to be equivalent to the respective entity in the electrical and software models.

Requirement 1.3 (Links between entities) — *Links between entities in disparate models must be captured, thereby explicitly defining the overlap between the models.*

3.1.2. Requirements Regarding the Support to Diagnose Inconsistencies

If a common knowledge base, which contains the models under investigation, is available (i.e., Requirement 1 is fulfilled), respective mechanisms to manage inconsistencies can be put in place. One essential property to be fulfilled by an inconsistency management approach, according to [NER00], is the *diagnosis of inconsistencies* (see Requirement 2). As described in Chapter 2, essential to inconsistency management is the incorporation of different types of inconsistencies that can occur during engineering of automated production systems. Especially as such inconsistencies can be specific to companies and/or projects, its extensibility towards identifying, locating and classifying company- or project-specific inconsistencies is required.

Requirement 2 (Inconsistency diagnosis) — *An approach for inconsistency management in the automated production systems domain must provide the (semi-)automated means to diagnose different types of inconsistencies; that is, it must allow for identifying, locating and classifying [NER00] company- and/or project-specific inconsistencies.*

As different types of inconsistencies can occur during engineering of automated production systems, it is essential that these can be *specified* by an inconsistency management approach (see Requirement 2.1). Hence, an appropriate formalism must be found to define the inconsistency pattern to be matched against the knowledge base as well as to characterize the respective inconsistency (e.g., by defining to which category the inconsistency belongs). By that, the different types of inconsistencies discussed in Chapter 2 can be captured.

Requirement 2.1 (Specify) — *The means to specify – that is, to define and characterize – different types of inconsistencies must be provided.*

For realizing the diagnosis of inconsistencies, an appropriate mechanism to allow for – according to [NER00] – *identifying, locating and classifying* (see Requirement 2.2) the inconsistency is required. Given that such a mechanism is available, the specified inconsistencies can be matched against the model knowledge base and, based on the characterization of the inconsistency, the inconsistent model elements can be identified, located and classified.

Requirement 2.2 (Identify, locate, classify) — *The means to (semi-)automatically identify, locate and classify the specified inconsistencies must be provided.*

3.1.3. Requirements Regarding the Support to Handle Inconsistencies

In case an inconsistency was identified in the set of models, respective actions to handle the inconsistency must be derived (see Requirement 3). Similarly to the diagnosis of inconsistencies (see Requirement 2), it is crucial to allow for specifying different types of handling actions in case an inconsistency is diagnosed [NER00]. These handling actions may, on the one hand, include predefined actions that result from the type of inconsistency diagnosed in the models (e.g., a violation of a naming convention can be resolved by renaming the entity according to the convention). On the other hand, the definition of custom actions is necessary to provide the user with the flexibility to specify user-defined handling actions.

Requirement 3 (Inconsistency handling) — *An approach for managing inconsistencies in the automated production systems domain must provide the (semi-)automated means to handle different types of inconsistencies; that is, it must allow for either ignoring, tolerating or resolving [NER00] different types of inconsistencies.*

Crucial element to handle inconsistencies is the *specification* of handling actions (see Requirement 3.1). Therefore, an appropriate formalism must be found to define the inconsistency handling action to be performed on the knowledge base as well as to characterize the respective action (e.g., by defining to which category the handling action belongs).

Requirement 3.1 (Specify) — *The means to specify – that is, to define and characterize – handling actions for different types of inconsistencies must be provided.*

Typical inconsistency handling actions involve *ignoring, tolerating or resolving* a detected inconsistency (see Requirement 3.2). Consequently, an appropriate mechanism must be found that allows for handling a diagnosed inconsistency. For traceability purposes, if an inconsistency is ignored or tolerated, it is essential to capture the reason for ignoring or tolerating the inconsistency as well as to document the stakeholder that decided to ignore or tolerate the inconsistency. For resolution strategies, it is, on the one hand, essential to predefine typical resolution actions for typical types of inconsistencies and, on the other hand, to allow users to execute user-specific resolutions.

Requirement 3.2 (Ignore, tolerate, resolve) — *The (semi-)automated means to ignore, tolerate or resolve the diagnosed inconsistencies must be provided.*

3.1.4. Requirements Regarding the Support to Measure and Assess Diagnosed Inconsistencies and Handling Actions

Assuming that an inconsistency management approach for the automated production systems domain is capable of *diagnosing* (cf. Requirement 2) and *handling* (cf. Requirement 3) inconsistencies, it is inevitable to provide users with the means to *measure and assess both the impact of a diagnosed inconsistency and the effort to handle the respective inconsistency* (see Requirement 4). This is especially essential, as a multitude of different inconsistencies can occur, with different degrees of severity and disparate impact on the quality of the engineering solution. For instance, whereas a violation of a naming convention (e.g., for a respective software variable) may be undesired, its impact on the quality of the overall solution can be neglected. However, if inconsistencies provide users (e.g., domain experts or discipline engineers) with potentially false information, erroneous decisions can be made. Therefore, it is essential to allow users to measure the impact of diagnosed inconsistencies as well as to asses the impact of potential handling actions.

Requirement 4 (Measurement and assessment capabilities) — *An approach for managing inconsistencies in the automated production systems domain must allow users to measure the impact of diagnosed inconsistencies and to assess the impact of respective handling actions.*

As discussed beforehand, the means to *measure inconsistencies* must be provided (see Requirement 4.1). For one, this includes the identification of the expected impact of an inconsistency. For instance, a warning (e.g., as a consequence of the violation of a naming convention) will have less severe impact on the overall quality of the engineering solution; an error (e.g., as a consequence of erroneous or contradicting information) could potentially lead to wrong decisions within the engineering process. Moreover, it is essential to conclude on the necessary stakeholders that are responsible to handle the diagnosed inconsistency. With this information, it can be determined who will need to decide upon inconsistency handling actions.

Requirement 4.1 (Measure) — *The means to measure the diagnosed inconsistencies must be provided. This includes (1) the identification of the expected impact of an inconsistency (e.g., inconsistency corresponds to an error, a warning or an information) as well as (2) the identification of the respective stakeholders that are responsible to handle the inconsistency.*

If an inconsistency is diagnosed, respective handling actions can be determined. However, in most cases it is not possible nor desirable to automate the execution of potential handling actions – rather, the responsible stakeholders must be able to decide, which handling action must be taken. Consequently, in order to support these stakeholders in their decision, it must be possible to *assess the impact of handling actions* (see Requirement 4.2) (e.g., by giving an estimation of the time to resolve an inconsistency). After having executed a handling action, stakeholders must be able to re-evaluate, whether the inconsistency has been handled appropriately or not.

Requirement 4.2 (Assess) — *The means to assess the impact of a handling action must be provided. This includes (1) to estimate the cost of a handling action (e.g., time to resolve the inconsistency) and (2) to re-evaluate whether the inconsistency has been appropriately handled or not.*

3.1.5. Requirements Regarding the Operationalization

Whereas the previous Requirements 1 to 4 focused on the technical perspective of an inconsistency management approach, it is essential to also incorporate the means to *operationalize* (see Requirement 5) the approach in the automated production systems domain. On the one hand, this includes the applicability of the approach in industrial settings of automated production systems, in which the appropriateness of the approach for, e.g., domain experts such as mechanical, electrical and software engineers, must be considered. On the other hand, extensions towards company-/project-specific requirements must be considered. Finally, an approach for inconsistency management depends on its scalability and performance, which are – especially for industry-scale systems that often consist of thousands of components – essential to real-world applications.

Requirement 5 (Operationalization) — *For an approach to manage inconsistencies in the automated production systems domain, the applicability for domain experts, i.e., the discipline-specific engineers participating in the engineering process, in industrial settings must be ensured.*

A first aspect of operationalization of the inconsistency management approach for industrial automated production systems is its *extensibility* (see Requirement 5.1), e.g. for project- or company-specific applications. On the one hand, such extensions require the means to incorporate project- or company-specific models, inconsistencies as well as handling actions. Hence, the mechanisms, concepts and tools used for inconsistency management must support the incorporation of such project- or company specific documents. On the other hand, it is essential to ensure that an inconsistency management approach can be adapted to project- or company specific tool chains and set-ups. It is only then that manufacturers in the automated production systems domain will be able to adapt and apply the approach for their purposes. As a consequence, the technologies being used by an inconsistency management approach must rely upon accepted standards.

Requirement 5.1 (Extensibility) — *The mechanisms and tools being used for inconsistency management must be extensible for project- or company-specific applications. This includes the means to adapt and extend the approach towards project- or company-specific models, inconsistencies as well as handling actions.*

A second aspect that refers to the operationalization of the inconsistency management approach is its *comprehensibility* for potential users (see Requirement 5.2). Potential users of such an approach include domain experts and discipline engineers. Therefore, the mechanisms and tools being used by these users must be appropriate for them.

Requirement 5.2 (Comprehensibility) — *The mechanisms and tools being used for inconsistency management must be comprehensible for users (e.g., for domain experts).*

Finally, the operationalization of an inconsistency management approach also demands its applicability for real-world industry-scale applications with regard to *scalability and performance* (see Requirement 5.3). As industry-scale systems often contain thousands of components and, hence, the models describing these system often comprise hundreds of thousands of entities, the mechanisms and tools being used for the purpose of inconsistency management must be scalable and with adequate performance to incorporate these kinds of systems.

Requirement 5.3 (Scalability and performance) — *The mechanisms and tools being used for inconsistency management must be applicable for industry-scale applications.*

3.2. Simplifying Assumptions

As discussed within the previous sections, manifold considerations must be made for the development of an inconsistency management approach in the automated production systems domain. To reduce the investigation to a manageable scope, a number of simplifying assumptions underlie this dissertation. These assumptions are as follows.

Assumption 1 (Focus on engineering) — *Conclusions are drawn from engineering information. Inconsistencies during operation of automated production systems are not considered.*

Assumption 2 (Focus on models) — *Conclusions are drawn from models, i.e., from the information content within discipline-specific engineering models. Textual, fuzzy information is not considered for inconsistency management.*

Assumption 3 (Focus on static data) — *Conclusions are drawn from static information and knowledge. It is not focused on executing, e.g., software or analysis models.*

Assumption 4 (Focus on technical realization) — *It is focused on the technical realization of an inconsistency management approach. Although part of the considerations throughout this dissertation (see Requirement 5), the means to introduce the inconsistency management approach in an industrial (project or company) setting goes beyond the scope of this dissertation.*

Assumption 5 (Neglect of managing variants and versions) — *It is focused on the current version and variant of the models under consideration. Although not limited to, variant and version management of models is neglected within the context of this dissertation.*

3.3. Summary

Within the preceding sections, the requirements and assumptions according to the different parts of an inconsistency management approach in the automated production systems domain are deducted. All of the requirements and assumptions were given short names, which will be referred to in the following chapters.

Chapter 4.

Related Work

This chapter compares the different approaches in the related literature, which provide concepts that are related to inconsistency management in the automated production systems domain. First, approaches are introduced that aim at integrating the heterogeneous models into a single language or format (Section 4.1). Second, approaches in the field of inconsistency management are discussed (Section 4.2). Finally, software tools that are available for the engineering of automated production systems are discussed regarding their feasibility for inconsistency management (Section 4.3). The chapter is concluded in Section 4.4.

4.1. Approaches to Integrate Heterogeneous Models

The increasing complexity in the automated production systems domain is often faced by means of introducing appropriate modelling languages that abstract the view on the technical system, thereby reducing complexity and providing the user with adequate means to rapidly comprehend the system under consideration. However, introducing modelling languages arises the need to cope with inconsistencies that potentially arise in between or within the models. In order to cope with these inconsistencies, mainly two approaches can be identified: For one, integrated modelling languages or formats are used, which aim at integrating the different models in the automated production systems domain into one overarching language, thereby reducing the potentially occurring inconsistencies. These *integrated modelling languages and formats* are discussed in Section 4.1.1. Moreover, *model mappings and links* are used to explicitly capture the dependencies between disparate models. These approaches are discussed in Section 4.1.2. An overview of the discussed approaches can be found in Tables 4.1 and 4.2; a conclusion of the findings is given in Section 4.1.3.

4.1.1. Integrated Modelling Languages and Formats

As discussed beforehand in Section 2.2, the use of models serves the purpose of abstraction, thereby focusing on certain points of interest in order to handle the complexity of the system under investigation. Consequently, Model-Based Engineering (MBE) approaches rapidly became the standard development paradigm for complex software by focusing on the (software) system instead of on the computing concepts and environments [Sch06] – hence, "[...] shifting much of the focus away from program code" [TF11]. By means of MBE, domain-specific modelling languages can be constructed through metamodels for the purpose of building applications that are tailored to the specific domain, as well as transformation engines and generators to analyse and synthesize certain modelling aspects [Sch06]. Several graphical modelling languages have, thus, been developed to provide an intuitive, graphical development process.

UML-based Modelling Approaches

For the software development, the Unified Modeling Language (UML) [OMG15c] has evolved as the de facto standard [SBF07] and, hence, its application for control software development seems obvious. UML pursues an object-oriented approach and allows for encapsulating and reusing entities by

Table 4.1. Overview of the related work in the field of integration of heterogeneous models: integrated modelling languages and formats

		Requirements											
		Req. 1			**Req. 2**		**Req. 3**		**Req. 4**		**Req. 5**		
Name	**Domain**	**Common syntax (1.1)**	**Common semantics (1.2)**	**Links between entities (1.3)**	**Specify (2.1)**	**Identify, locate, classify (2.2)**	**Specify (3.1)**	**Ignore, tolerate, resolve (3.2)**	**Measure (4.1)**	**Assess (4.2)**	**Extensibility (5.1)**	**Comprehensibility (5.2)**	**Scalability and performance (5.3)**
Integrated Modelling Languages and Formats (Section 4.1.1)													
[BGT05; SW10; GS13]	Mechatronics	●	●	◐	◐	◐	○	○	n/a	n/a	●	●	n/a
[Bas$^+$11; BFB14; SBF07]	Manufacturing	●	◐	◐	◐	◐	○	○	n/a	n/a	●	●	◐
[Thr05; Thr10; Thr13]	Mechatronics	●	◐	◐	◐	◐	○	○	n/a	n/a	●	●	n/a
[KV13; Ker$^+$13; Ker$^+$14]	Mechatronics	●	◐	◐	○	○	○	○	n/a	n/a	●	●	n/a
[SSP09; CLP11; Sha$^+$12]	Systems	●	●	●	◐	◐	○	○	n/a	n/a	●	●	◐
[KP09; KP10]	Systems	●	●	○	○	○	○	○	n/a	n/a	●	●	n/a
[Lin$^+$15a; Lin$^+$15b]	Production	◐	◐	○	◐	●	◐	○	◐	n/a	●	●	n/a
[EMO07; Mar$^+$09]	Control	●	◐	◐	◐	◐	○	○	n/a	n/a	●	●	◐
[Dra10; IEC14]	Control	●	◐	◐	n/a	n/a	n/a	n/a	n/a	n/a	●	●	◐
[EM12]	Control	●	●	◐	◐	◐	○	○	n/a	n/a	●	●	◐
[Ber$^+$16]	Production	●	◐	○	○	○	○	○	n/a	n/a	●	●	◐
[Car$^+$16]	Production	●	◐	◐	○	○	○	○	n/a	n/a	●	●	◐
[Fay$^+$17]	Production	●	●	◐	○	○	○	○	n/a	n/a	●	●	○
[BSZ09; Mos$^+$11; MB12]	Production	●	●	○	◐	n/a	◐	n/a	n/a	n/a	●	●	◐
[Mor$^+$12; Bif$^+$14b]	Production	●	●	◐	○	◐	○	◐	n/a	n/a	●	●	◐
[MSD10]	Production	●	●	◐	○	○	○	○	n/a	n/a	●	◐	●
[Eka$^+$17]	Systems	●	●	●	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Legend: ● – in focus of this work, ◐ – partially in focus of this work, ○ – not in focus of this work, n/a – not applicable means of classes. The value in applying UML for complex software systems especially is in improved comprehensibility and, thus, higher reliability and flexibility in control software system design and maintenance [SBF07; TF11]. Although object-orientation is not yet state of the art in industrial practice – even if an object-oriented extension of IEC 61131-3 has recently been proposed – control software developers are already familiar with the general concept of object-orientation [Wer09] owing to the concept of Function Blocks (FBs) instantiation. In addition, experiments have shown that – although imposing several restrictions on teaching of UML compared to IEC 61131-3 [Vog+13] – UML can increase comprehensibility and flexibility for control software development [Vog15]. Model-to-model transformations as well as model-to-text transformations for the purpose of generating control software from UML diagrams have already been developed [WV11]. In addition, design patterns were defined to further ease the development of control software [BFS13]. However, although UML is appropriate for representing the control software discipline of an automated production system, there is no inherent support for capturing other discipline aspects, e.g., from the electrical or mechanical engineering discipline (cf. Requirement 1).

In order to integrate such additional aspects into MBE, the UML provides the means to extend existing diagrams by means of so-called profiles. Several profiles have been proposed to incorporate diverse aspects, e.g., UML-RT [Sel98] for incorporating real-time aspects or MARTE [OMG11] for model-driven development of Real-time Embedded Systems. However, both UML-RT and MARTE focus on the software aspect of the automated production system and do not incorporate aspects from other disciplines such as mechanical engineering or electrical engineering (cf. Requirement 1). In Burmester et al. [BGT05], MechatronicUML is proposed, which focuses on the real-time behaviour of complex, distributed systems. Therein, both a structural view and a behavioural view are provided, which allow to flexibly define the reusable components of a mechatronic system. Applications of MechatronicUML were illustrated, e.g., in the railway domain [SW10; GS13] with a special focus on verification of self-optimizing mechatronic systems. Although both a common syntax and semantics for mechatronic systems is enabled by means of a UML profile (cf. Requirement 1), inconsistency management is not in focus of these works (cf. Requirements 2 to 4). In Secchi et al. [SBF07], the UML-RT is applied together with bond graphs in order to describe the structure and behaviour of both the control software and the physical system. However, although these profiles incorporate the interdisciplinary aspects that are necessary for the respective application, the different engineering documents from the different disciplines are not yet included in these languages (cf. Requirement 1).

SysML-based Modelling Approaches

Contrary to MBE for the software perspective of automated production systems, Model-Based Systems Engineering (MBSE) constitutes the shift from a document-centric development process of systems towards the usage of integrated models for requirements, design, analysis, verification and validation [DM13]. Therein, the Systems Modeling Language (SysML) [OMG15b] has become one of the most wide-spread languages and extends the software-centric UML towards are more system-centric, graphical modelling language. SysML is "[a]n important extension of the UML [...] for systems engineering" and "adds stereotypes to the UML as well as new diagram types [...] while leaving out some UML diagrams" [RP11] for the purpose of systems engineering.

Various research work has been carried out on MBSE using SysML, putting the focus on different modelling purposes and development phases. Bassi et al. [Bas+11] as well as Barbieri et al. [BFB14] aim at an integrated modelling process, in which "SysML models are adopted as high-level abstractions of the system under design" [Bas+11]. By that, a hierarchy of models is created to support three levels of modelling [Bas+11] – including a *detail level* (e.g., domain-specific models to describe internal system dynamics), a *global level* (i.e., a model that describes the global system dynamics) and a *high-level model* for the purpose of management between the involved models. Similarly, Barbieri et al. [BFB14] propose a hierarchical approach to link the conceptual system design with executable simulation models. The resulting model hierarchy provides a high degree of flexibility as modellers can integrate their domain-specific modelling languages into a common SysML-based syntax (cf. Requirement 1.1) – the semantic concepts used within the different modelling languages as well as the links between these concepts are, however, not described formally and limited to the expressiveness of the SysML syntax and semantics (cf. Requirements 1.2 and 1.3). Although the intended development process also involves a *validation phase* to avoid errors and inconsistencies during system design, these checks are limited to simple syntactic validation checks and value comparisons. A sophisticated specification, diagnosis and handling of complex inconsistencies is not yet supported (cf. Requirements 2 and 3).

Model-Integrated-Mechatronics [Thr05] introduces a component-based approach, in which different *layers* are used in order to describe the mechatronic components. Of essential importance to the overall concept is the *mechatronic layer*, which is "projected into three dimensions representing the *application* [layers], the *resource* [layer] and the *mechanical* [layer] [...]" [Thr05]. By that, engineers combine mechatronic components in the mechatronic layer including their already existing descriptions in the detailed layers. Besides the segmentation into vertical *layers*, Model-Integrated-Mechatronics also supports model evolution as it divides the engineering process into *analysis*, *design* and *implementation*. Therein, Model-Integrated-Mechatronics foresees a flexible development process, in which engineers can work both "horizontally and vertically, either top-down or bottom-up" [Thr10]. By means of SysML, mechatronic components can be modelled and combined according to the Model-Integrated-Mechatronics paradigm [Thr05]. Consequently, a SysML model serves as the basic, synchronizing model, which provides the respective links to further, domain-specific models, e.g., in the application, resource or mechanical layer. By relying on a common syntax and semantics provided by the proposed SysML 3+1 view-model, the knowledge base is formed in an unambiguous manner (cf. Requirement 1). Nevertheless, the diagnosis and handling of inconsistencies in this work is limited to syntactical compliance checks to the central SysML view, which supports, e.g., traceability between the different modelling artefacts [Thr13] – the means to specify, diagnose and resolve user-specific inconsistencies as demanded in Requirements 2 and 3 is not supported.

SysML4*Mechatronics* is the modelling language introduced by Kernschmidt et al. [KV13]. The authors extend the SysML metamodel using a profile for the purpose of modelling discipline-specific components of the system under development as well as discipline-specific ports in between these components. By means of SysML4*Mechatronics*, an increased reuse of system components is envisioned, as mechatronic modules can be combined out of discipline-specific components. Different mechanisms have been proposed in order to extend the modelling language, e.g., for the purpose of analysing change situations [Ker+14] as well as for integrating SysML4*Mechatronics* with further modelling languages [Ker+13] within a common syntax (Requirement 1.1). Nevertheless, although a comprehensive, interactive visualization of the system under investigation can be achieved by SysML4*Mechatronics* (Requirement 5), diagnosis and handling of inconsistencies within heterogeneous models is not in focus of their work (Requirements 2 and 3).

A similar approach that aims at applying SysML as a unifying language is presented in Shah et al. [SSP09]. Therein, at the example of two views – namely a system-level view in SysML and a domain-specific electrical view in EPLAN P8 – the authors illustrate their mapping approach. Further examples for integrating SysML with domain-specific models can be found, e.g., for the purpose of simulation [CLP11] and optimization [Sha+12]. Domain-specific metamodels are, in a first step, created to formally define the involved domains. Subsequently, the SysML metamodel is customized by means of SysML profiles to allow for domain-specific modelling. Finally, the views are integrated by means of a mapping between the domain-specific metamodel and the SysML profile. By that, contrary to implementing a multitude of bidirectional mappings between domain-specific languages, SysML can serve as common, multi-purpose language to capture the dependencies between different domains in a common system view [Sha$^+$10]. Consequently, the common concepts are captured in a SysML-based syntax and semantics as demanded in Requirements 1.1 and 1.2 and dependencies between the different views are considered explicitly (see Requirement 1.3) and in an extensible and comprehensible manner (Requirement 5). However, "a major research question involves determining an effective way [to] maintain consistency [...]" [Sha$^+$10]. As a consequence, further research is needed for the purpose of diagnosing and handling inconsistencies that are likely to occur between the different domain-specific models (Requirements 2 and 3).

Kerzhner et al. [KP10] make use of SysML for the purpose of model management. Three essential parts are introduced in a SysML model to define the systems engineering problems: *requirements* that should be met by the system, *experiments* to analyse, whether the requirements are fulfilled, and *system topologies* that are potential candidates to fulfil the requirements. By that, a first step towards an automatic search for valid and appropriate system alternatives as suggested in Kerzhner et al. [KP09] is provided. The central SysML model allows for capturing the information that is needed for evaluation of design alternatives (Requirement 1). However, inconsistency management between the different models involved during engineering is not in the focus of this work (cf. Requirements 2 and 3).

Lin et al. [Lin$^+$15a] present a SysML-based approach for change request management. Therein, SysML diagrams are used to support the decision management in case changes are made to the automated production system. Therein, a workflow is proposed that allows for identifying the model elements to be replaced, integrating the novel elements to the engineering solution and identifying the model elements that are affected by the change request. Besides the workflow, a technological basis is provided by means of a SysML profile [Lin$^+$15b]. Although the presented workflow and modelling support aids engineers in managing change requests to the engineering solution, it is not focused on managing inter-model inconsistencies (cf. Requirements 2 and 3).

Exchange Formats

To address the challenge of interoperability, several approaches aim at providing respective exchange formats that allow for a generic description of the system architecture. For one, Estévez et al. [EMO07] introduce a generic markup language for defining both the software and hardware architecture of industrial control systems. By means of the markup language, two essential benefits can be obtained: First, consistency between hardware and software architectures can be checked by means of an Extensible Markup Language (XML) schema as well as XML Schematron rules. Second, the IEC 61131-3-compliant automation projects can be generated. The authors argue that the main benefit in using XML is in the power of additional XML technologies. Besides validating the compliance to lexical and syntactical constraints defined in the XML schema, a multitude of technologies such as Extensible Stylesheet Language (XSL) Transformation (XSLT) as well as respective standardized interfaces to generate or manipulate XML documents exists [Mar$^+$09]. As a consequence, the proposed markup language captures the necessary elements from software and hardware architectures (cf. Requirement 1) to ensure simple syntactical and lexical constraints (cf. Requirement 2) – however, more complex, user-defined inconsistencies cannot be specified and resolved by the approach (cf. Requirements 2 and 3), making additional support for specification, diagnosis and handling of user-defined inconsistencies necessary.

One upcoming standard in the automated production systems domain is Automation Markup Language (AutomationML), which aims at "support[ing] the data exchange in a heterogeneous engineering tools landscape" [IEC14]. Therein, the objective of AutomationML is to interconnect different discipline-specific engineering tools such as Programmable Logic Controller (PLC) programming tools, mechanical engineering tools, electrical engineering tools, etc. [Dra10; IEC14] AutomationML makes use of Computer Aided Engineering Exchange (CAEX) [IEC16] as the basis to connect the different data formats – hence, CAEX serves as the top-level format being used within AutomationML. Discipline-specific XML formats – among the Collaborative Design Activity (COLLADA) and PLCopen XML [PLC09b] format, which are currently included in AutomationML, further discipline-specific XML formats can be used – are then interconnected by means of the CAEX capabilities. By that, tool vendors can export their tool-specific documents into the AutomationML format, thereby providing the basis for engineering data exchange between different discipline-specific tools and formats. Although AutomationML gains more and more importance in various fields in the automated production systems domain and appropriate semantic mappings can be represented in AutomationML [Bif$^+$14a], sophisticated support for linking of engineering artefacts (cf. Requirement 1) is not available inherently [Bif$^+$15] – but, nonetheless, an essential prerequisite for inconsistency management (cf. Requirements 2 and 3).

As a basis to overcome this drawback, Estévez et al. [EM12] suggest the use of AutomationML combined with Mathematical Markup Language (MathML) [W3C14a] – a markup language to encode mathematical content – to ensure the correctness of AutomationML documents. By that, design errors can be identified in the syntax and static semantics of the respective documents. Although these simple validations are essential to provide correct models to engineers working with the documents, complex semantic inconsistencies cannot be described easily (cf. Requirements 1 and 2). Furthermore, support for handling identified inconsistencies (cf. Requirement 3) is not in focus of this work.

Within the approach presented by Berardinelli et al. [Ber$^+$16], an integration between AutomationML and SysML is envisioned, thereby bridging the gap between exchange formats and model-driven engineering technologies. By such an integration, the SysML model serves as visual representation of the knowledge contained within AutomationML. To realize such a transformation, the authors formulate a SysML profile, which allows for extending the SysML modelling constructs towards AutomationML-specific ones. By means of a model-to-model transformation formulated using the ATLAS Transformation Language (ATL) [Jou$^+$08], the transformation between the SysML profile and the AutomationML files is specified. As a consequence, the data modelled in the SysML model can be transformed into the standardized syntax and semantics of AutomationML (Requirements 1.1 and 1.2). The benefit in applying such transformations between a priori decoupled technologies is in using the benefits of both – for instance, Kovalenko et al. [Kov$^+$15] identify the benefit in transforming between AutomationML and Web Ontology Language (OWL) for the purpose of reasoning and querying on the AutomationML data. However, neither of the proposed approaches aims at the management of inconsistencies in between different, heterogeneous engineering models (Requirements 2 and 3).

The approach presented by Carlsson et al. [Car$^+$16] proposes a plant description format for increased interoperability of engineering tools. By means of this approach, an exchange format is proposed "for the exchange of engineering data which does not force all systems to use the same standard or required full compatibility between all relevant standards" [Car$^+$16]. Rather, a basic structure is provided to represent plant descriptions from different engineering tools and formats. However, although an essential knowledge base is created by means of the presented format (Requirements 1.1 and 1.2), inconsistency management is not in focus of their work (Requirements 2 and 3).

Within the research project SemAnz4.0, a concept for modelling technical systems in a model- and lifecycle-spanning manner is created [Fay$^+$17]. Therein, during engineering, a holistic system information model is created by means of a *system* and a *property model*. The *system model* represents the technical system from the different discipline-specific modelling perspective. By means of the *property model*, system properties can be described in a machine-interpretable manner for further (semi-)automatic interpretation. For both models, standardized metamodels are envisioned.

Data and information hubs

As one means to simplify the interdisciplinary engineering, some approaches aim at providing a central information system, which captures the heterogeneous engineering data in a common representation. By that, all participating stakeholders and disciplines can work on this central data or information hub.

An example for such an information hub is the Automation Service Bus [BSZ09], which aims at systematically integrating systems engineering software tools by means of common concepts within an Engineering Knowledge Base [Mos^{+}11; MB12]. According to Moser et al. [MB12], typical application examples of such a hub within the automation systems domain are, e.g., tool-crossing data exchange, consistency checking as well as end-to-end analyses. Although, with the conceptual approach presented in Moser et al. [MB12], an essential basis for inconsistency management is laid – namely, a common syntax and semantics for integrating different engineering models (Requirements 1.1 and 1.2), the authors do not specify, how the actual inconsistency management process should be realized and integrated (Requirements 2 and 3).

Such an Automation Service Bus can serve as the basis to realize a multitude of use cases – e.g., dashboards [Bif^{+}14b] to aid in identifying and tracing interdisciplinary dependencies and parameters (Requirement 1.3) as well as interactive navigators [Mor^{+}12] to support navigation across different engineering tools (Requirement 5.2). Moreover, the applicability of the Automation Service Bus for synchronization processes was successfully validated by comparing the approach with manual synchronization processes [WB15], therein especially focusing on a comprehensible synchronization process for identifying defects between the engineering data (Requirement 2) and supporting their resolution (Requirement 3).

A concept for a semantic integration of disparate models within the domain of mechatronic systems is presented by Muehlhause and Diedrich [MSD10]. Therein, Semantic Web Technologies are used to represent the models in a common representational formalism with a common semantics (Requirement 1) – however, inconsistency management (Requirements 2 and 3) is not in focus of their work.

A comprehensive survey of ontology-based data integration approaches can be found in [Eka^{+}17]. Therein, the authors argue that 4 stereotypes of such approaches can be found. For one, the authors distinguish between *single-ontology* approaches and *multiple-ontology* approaches. Single-ontology approaches assume one single global vocabulary to integrate all data sources. Multiple-ontology approaches allow for multiple local ontologies being used for different data sources and respective semantic mappings to integrate them. A *hybrid ontology* approach allows for a shared vocabulary to be used and extended by the different data sources. Finally, a *global-as-view* approach assumes a global ontology with no extension through local ontologies. Consequently, instead of the need to redefine local ontologies (as is the case for the hybrid ontology approach), local ontology definitions can be preserved. Together with this classification, Ekaputra et al. [Eka^{+}17] propose a decision tree to select for the best approach based on the concrete use case. Nevertheless, a concrete implementation framework or guidance is not in scope of their work (Requirement 5).

4.1.2. Model Mappings and Linking Support

As discussed beforehand, an essential basis for inconsistency management is to establish and formally capture the links between overlapping model entities (see Requirement 1.3). Several approaches have been developed in the related literature that focus on these mapping between modelling languages and formally capture the links between semantically overlapping model entities.

As a basis for different purposes, e.g., (in-)consistency management, versioning, tool integration and transformation, Qamar et al. [Qam^{+}12] analyse the need to formally model dependencies (that is: semantic overlaps or links) between different model entities. As a basis for their work, they distinguish between so-called *Synthesis Dependencies*, which represent the choice made by an engineer or parametric function within an engineering problem, and *Analysis Dependencies*, which represent mathematical relationships between a set of dependencies. By means of these dependencies, the authors propose the formulation of a network of dependencies as a basis for, e.g., inconsistency management [Qam+12]. To allow for formulating these dependencies for modelling languages such as SysML, the authors make use of a Domain-specific Language (DSL) for dependency modelling. By means of this graphical language, for one, links can be captured between different model entities. Moreover, the network of dependencies can, e.g., be visualized as a graph. A prototypical implementation of the suggested dependency modelling language is presented in Qamar et al. [QWD15]. Although dependency modelling by means of a DSL is an essential basis for inconsistency management (see Requirements 1.1 and 1.3), the semantics of these dependencies must be captured (Requirement 1.2) in order to (semi-)automatically check whether the dependencies are held or not, and to (semi-)automatically propagate changes along these dependencies (Requirements 2 and 3).

An approach that envisions to formally capture so-called *engineering effect links* – that is, relations, which result from, e.g., decisions during the engineering workflow – is introduced in Schröck et al. [Sch+13a]. Therein, the authors aim at an engineering process that (1) captures a set of viewpoints covered throughout the engineering as well as (2) allows for establishing such engineering effect links throughout the engineering of a machine or plant. Later, the authors evolve the term effect link towards *engineering relations*, which allow to capture different classes of relations among engineering disciplines [Sch+15]. As a consequence, using these engineering relations, a multitude of views on the system under development (e.g., a functional view and a variant view) can be combined. The engineering relations then not only serve as the basis to ensure consistency among the different views in a formal way, but also to support engineers in reusing discipline-specific documents [SFJ15]. A prototypical tool support is provided by Schröck et al. [Sch+15] at the example of pure::variants [pur16] and COMOS [Sie16]. However, although the authors discuss the need to provide a standardized way to capture these engineering relations, neither a formalism to capture these links as well as the linked model entities (Requirement 1.1) nor the means to formally capture the (static and operational) semantics of links (Requirement 1.2) are proposed. These are yet to be defined, especially for inconsistency management for heterogeneous models in the automated production systems domain.

The need to support links between AutomationML files and engineering data is highlighted in Biffl et al. [Bif+14a; Bif+15]. Therein, Biffl et al. [Bif+15] formulate an AutomationML metamodel as the basis to provide tool support for this purpose. Links between disparate AutomationML files are then represented in a model-based manner. Furthermore, the authors make use of the Object Constraint Language (OCL) for the purpose of formulating and executing simple inconsistency rules on the link model. Tool support is provided by means of the Eclipse Modeling Framework (EMF) [Ecl16c] as well as the Eclipse Epsilon Framework [Ecl16b], which together provide the means to formulate respective metamodels for AutomationML as well as for links in between these models. Although by means of the proposed concept, an initial support towards linking AutomationML with engineering data is provided in a common syntax (Requirements 1.1 and 1.3), future research needs to investigate usable and comprehensible means, e.g., for inconsistency management [Bif+15] – and hence, for supporting the diagnosis and handling of inconsistencies (Requirements 2 and 3).

Lüder et al. [LPW18] present such an approach that makes use of OCL for the purpose of checking inconsistencies on links within AutomationML. Within their research, a conceptual approach is introduced, which is yet subject to further research for practical applications (Requirement 5).

For the domain of product-service systems, an ontological approach to couple the different disciplines is presented in Zou et al. [Zou+19]. Therein, by means of ontological background knowledge (Requirement 1), a basis for the means of inconsistency management can be formed. Although analogously, Kattner et al. [Kat+19] introduced an approach to capture the dependencies between the different disciplines explicitly, a standardized and common mechanism to define and measure inconsistencies (Requirement 2) or handling actions (Requirement 3) has not been discussed.

Konersmann et al. [KG12; Kon18] propose an approach for explicitly integrating architectural software models with program code to ensure consistency between both. Although the approach proposes an intermediate language to capture the modelled/programmed knowledge (Requirement 1) as well as well-defined transformations that ensure that both models and program code are free of inconsistencies (Requirement 2), the authors put their focus on the domain of software systems. Hence, the interdisciplinary engineering models of automated production systems are not in focus of their work (Requirement 5).

Table 4.2. Overview of the related work in the field of integration of heterogeneous models: model mappings and linking support

		Requirements											
		Req. 1			**Req. 2**		**Req. 3**		**Req. 4**		**Req. 5**		
Name	**Domain**	**Common syntax (1.1)**	**Common semantics (1.2)**	**Links between entities (1.3)**	**Specify (2.1)**	**Identify, locate, classify (2.2)**	**Specify (3.1)**	**Ignore, tolerate, resolve (3.2)**	**Measure (4.1)**	**Assess (4.2)**	**Extensibility (5.1)**	**Comprehensibility (5.2)**	**Scalability and performance (5.3)**
Model Mappings and Linking Support (Section 4.1.2)													
[Qam⁺12; QWD15]	Mechatronics	●	◐	●	○	○	○	○	n/a	n/a	●	◐	◐
[Sch⁺13a; Sch⁺15; SFJ15]	Production	◐	◐	●	○	○	○	○	n/a	n/a	◐	◐	●
[Bif⁺14a; Bif⁺15]	Production	●	◐	●	◐	○	○	○	n/a	n/a	●	◐	◐
[LPW18]	Production	●	◐	●	◐	○	○	○	n/a	n/a	●	n/a	n/a
[Zou⁺19; Kat⁺19]	Product-service systems	●	●	●	◐	○	○	○	○	○	◐	n/a	n/a
[KG12; Kon18]	Software	●	◐	●	◐	○	◐	○	n/a	n/a	◐	n/a	n/a

Legend: ● – in focus of this work, ◐ – partially in focus of this work, ○ – not in focus of this work, n/a – not applicable

4.1.3. Synopsis

A multitude of modelling languages – e.g., UML for the control software development and SysML for a systems engineering perspective on the automated production systems – is available for the purpose of comprehensibly and extensibly model the knowledge on the system under investigation (cf. Requirements 1 and 5). Furthermore, AutomationML is an upcoming standard that gains more and more importance for exchanging engineering data between different discipline-specific engineering tools. However, although some approaches to diagnose simple syntactical inconsistencies exist, neither the aforementioned modelling languages nor standards such as AutomationML inherently support the means to diagnose and handle complex inconsistencies (cf. Requirements 2 and 3). Therein, especially the description of the common semantics of modelled entities as well as the links between these entities (Requirements 1.2 and 1.3) are essential for inconsistency management.

4.2. Approaches to (Semi-)Automated Inconsistency Management

Whereas the previously discussed research works focus on the integration of heterogeneous modelling languages and formats, it is essential to ensure that these heterogeneous models and documents are free of inconsistencies. Consequently, this section is dedicated to inconsistency management approaches. These approaches can broadly be classified into three different categories: Approaches that make use of *logical reasoning and theorem proving* aim at a formal knowledge base in terms of a set of axioms, from which the (in-)consistency of a set of models can be formally concluded (Section 4.2.1). Within *rule- and pattern-based inconsistency management* approaches, (in-)consistency rules and patterns are used to describe the circumstances under which a model is (in-)consistent (Section 4.2.2). By means of *model synchronization* approaches, it is aimed at avoiding inconsistencies through synchronizations between the models under investigation (Section 4.2.3). These different approaches are compared in Table 4.3 – the findings are concluded in Section 4.2.4.

4.2.1. 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.

Finkelstein et al. [Fin+94] make use of first-order logic to diagnose inconsistencies within multi-view software models such as class diagrams, sequence diagrams, etc. In order to capture the links between the different entities within the models, they manually add statements to the models. Whether an inconsistency occurs or not is inferred by an automated theorem prover. By means of domain-specific rules, which are specified in temporal logic, inconsistencies can be resolved. By that, the approach of Finkelstein et al. [Fin+94] allows for specifying, diagnosing and resolving inconsistencies within software models as demanded in Requirements 2 and 3 – however, heterogeneous models of automated production systems are not in focus of their work (cf. Requirement 1). Moreover, the use of first-order logic formalism hampers the potential application of the approach due to insufficient availability of respective experts in the field of automated production systems (Requirement 5).

Within the work of Schätz et al. [Sch+03], a first-order propositional logic similar to OCL is used to formulate consistency conditions and, by that, to describe the underlying formal model. Therein, they argue that consistency conditions exist at three distinct levels – namely, an *invariant conceptual level*, in which consistency conditions hold invariantly during the development, a *variant conceptual level*, which may be relaxed during certain development phases, and a *semantic* level. The latter refer to those consistency conditions, which result due to abstractions between the different models – e.g., *vertically* in case different granularities are described by the models or *horizontally* in case different, but overlapping aspects of the system are described. Although disparate aspects of the entire development process are focussed on by Schätz et al. [Sch+03], their approach is restricted to their underlying formal modelling language in the domain of embedded systems – further, discipline-specific languages and formats are not yet supported (Requirement 1). Moreover, measuring and assessing (Requirement 4) as well as handling inconsistencies (Requirement 3) is not in focus of their investigations.

UML models, in particular class diagrams, state charts and sequence diagrams, are in focus of Van Der Straeten et al. [Van+03]. In their work, a UML profile is introduced to express both consistency between models within the same version (*horizontal consistency*) and between different versions of the same model (*evolution consistency*). By means of Description Logics (DLs), different types of consistency can be checked within the UML profile, e.g., structural inconsistencies between class, sequence and class diagrams such as classless instances or dangling references as well as incompatible behaviour definitions between state charts and sequence diagrams. A rule-based inconsistency resolution approach can, based on the identified inconsistencies, be used to support

Table 4.3. Overview of the related work in the field of inconsistency management

		Requirements											
		Req. 1			**Req. 2**		**Req. 3**		**Req. 4**		**Req. 5**		
Name	**Domain**	**Common syntax (1.1)**	**Common semantics (1.2)**	**Links between entities (1.3)**	**Specify (2.1)**	**Identify, locate, classify (2.2)**	**Specify (3.1)**	**Ignore, tolerate, resolve (3.2)**	**Measure (4.1)**	**Assess (4.2)**	**Extensibility (5.1)**	**Comprehensi-bility (5.2)**	**Scalability and performance (5.3)**
Logical Reasoning and Theorem Proving (Section 4.2.1)													
[Fin^{+}94]	Software	◐	◐	◐	●	◐	◐	○	n/a	n/a	◐	○	n/a
[Sch^{+}03]	Embedded	◐	◐	◐	●	◐	○	○	○	○	◐	◐	n/a
[Van^{+}03; VD06; MVS05]	Software	◐	◐	◐	●	◐	●	◐	○	○	◐	○	n/a
Rule- and Pattern-Based Inconsistency Management (Section 4.2.2)													
[Egy11; Egy^{+}18; RE12; Dem^{+}16]	Software	◐	◐	n/a	●	●	●	○	◐	○	◐	●	●
[HEZ10]	Mechatronics	●	◐	◐	●	●	○	○	○	n/a	●	●	n/a
[MSD06; Van^{+}03]	Software	●	◐	◐	●	◐	●	◐	◐	●	●	◐	n/a
[Heg^{+}11]	Software	●	◐	○	●	●	●	◐	n/a	●	●	◐	n/a
[HP14; HQP14; Her15]	Systems	●	◐	●	●	◐	○	○	○	n/a	●	◐	n/a
[Kov^{+}14a; Kov^{+}14b; Kov^{+}15]	Production	●	●	◐	●	◐	○	○	◐	n/a	●	●	●
[Wol^{+}18; Wol19]	Product-service systems	●	◐	●	○	n/a	○	n/a	n/a	n/a	●	●	n/a
[Abe^{+}13]	Production	●	●	○	●	◐	○	○	○	n/a	●	○	○
[RF11; Gla^{+}15; GF16]	Production	●	◐	○	◐	○	○	○	n/a	n/a	●	●	◐
[Ahm^{+}17]	Assembly	●	◐	◐	●	◐	○	○	○	n/a	●	○	◐
[ZV17; ZLV18]	Production	n/a	n/a	◐	◐	◐	●	n/a	●	●	◐	○	●
Model Synchronizations (Section 4.2.3)													
[GW06; GW09]	Software	●	○	●	●	◐	●	◐	○	○	●	○	●
[Gau^{+}07; Gau^{+}09; Rie^{+}12]	Mechatronics	●	◐	●	●	◐	●	◐	○	○	●	◐	●
[KBL13; Kra^{+}15; Kra17; ABS18; Ana^{+}18]	Software	◐	◐	●	◐	◐	◐	◐	○	◐	●	◐	●

Legend: ● – in focus of this work, ◐ – partially in focus of this work, ○ – not in focus of this work, n/a – not applicable for (semi-)automatic resolution of inconsistencies [VD06; MVS05]. Whereas both diagnosing and handling inconsistencies is addressed by the authors (Requirements 2 and 3), their work is currently focussed on UML (software) models. Consequently, the extensibility of their work towards heterogeneous models of automated production systems (Requirements 1 and 5) – especially with regard to comprehensiveness of a DL-based approach for industry experts – needs to be evaluated in future works.

4.2.2. Rule- and Pattern-Based Inconsistency Management

Similarly to approaches that make use of logical reasoning and theorem proving, rule-based inconsistency management aims at applying a rule base that describes either the sufficient conditions that a model must satisfy for it to be considered consistent [HEZ10] – that is, rules are used as *positive constraints* – or as *negative constraints*, which represent the sufficient conditions that indicate an inconsistency [HQP14]. 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-based approaches is always incomplete [NER00]: Rules can be added and/ or removed without the need to rethink (and – in a worst case – adapt) the entire knowledge base.

Positive Constraints

Egyed et al. [Egy11] aim at diagnosing and tracking inconsistencies within software engineering models. What makes this approach unique compared to other inconsistency management approaches is that the authors do not restrict their inconsistency management to certain languages for defining and executing inconsistency diagnosis rules. Rather, they allow for any language to be used – the types of inconsistencies that can be identified depend on the language being used, more specifically on the consistency rule evaluator that needs to be implemented for the language. Furthermore, the authors argue that “[i]nstant feedback [...] is a fundamental best practice in the software engineering process” – as a consequence, they aim at incrementally checking the consistency of the models under consideration. As an example to exemplify their approach and to validate its scalability and performance, a multitude of UML models have been used – ranging from small models to large, industry-scale models. In addition, their approach has been "[successfully] applied [...] to models with more than 100,000 artefacts [...]" [Egy$^+$18]. Moreover, initial augmentation mechanisms were developed to repair inconsistencies [RE12]. However, although first efforts were made to integrate the approach with industrial tool suited to check for inconsistencies with electrical and software models [Dem$^+$16], no method for formulating consistency rules for heterogeneous models of automated production systems is given (Requirement 1). To overcome this drawback, mechatronic system models are focused on in Hehenberger et al. [HEZ10]. By means of a unifying and domain-spanning mechatronic ontology, model elements are tagged to implicitly specify the links between different models (Requirement 1.3) in a common, well-defined syntax and semantics (Requirements 1.1 and 1.2). Whereas the approach has been validated for large-scale software design models in Egyed [Egy11; Egy$^+$18], the appropriateness for industrial automated production systems engineering has not yet been investigated (Requirement 5).

Negative Constraints

One representative for a rule-based inconsistency management approach that makes use of negative constraints is the work of Mens et al. [MSD06]. The authors investigate consistency of UML class diagrams and protocol state machines, therein analysing mainly structural inconsistencies such as dangling type references, classless instances as well as instantiations of abstract entities. Contrary to their earlier works, in which DLs have been used for the purpose of managing behavioural inconsistencies [Van$^+$03], graph transformation rules are used to formulate inconsistency diagnosis and handling rules. By means of critical pair analysis, Mens et al. [MSD06] 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. Although restricted to the domain of software engineering, more precisely to UML class diagrams and protocol state machines (cf. Requirement 1), the presented approach provides an adequate basis to diagnose and handle a multitude of inconsistencies (Requirements 2 and 3). Besides, the critical pair analysis allows for pre-selecting a subset of existing resolution rules that reduce the available inconsistency resolutions to meaningful ones (Requirement 4). However, the applicability of the presented approach to more complex, industrial settings of heterogeneous models of automated production systems needs to be evaluated (Requirement 5).

Pattern-based Approaches

Within Hegedüs et al. [Heg$^+$11], so-called *quick fixes* are suggested as the means to diagnose and resolve inconsistencies in domain-specific languages. Quick fixes originate from the domain of software engineering, and are "[...] a very popular feature of integrated development environments [...], which aid programmers in quickly repairing problematic source code segments" [Heg$^+$11]. Hence, the authors aim at identifying inconsistencies within models by defining graph patterns, which are matched against the respective models. Additionally, graph transformations are used to specify actions to be taken in case an inconsistency is diagnosed. These actions are then visualized to the user as *quick fixes* – the best of which have been prioritized by means of a state-space exploration approach. In their work, Hegedüs et al. [Heg$^+$11] exemplify their approach by means of Business Process Model and Notation (BPMN) – a graphical notation for specifying business process models – and integrate their concept within the Viatra2 framework [BV06]. Nevertheless, applications to further modelling languages are enabled through the generic representation of inconsistency patterns and handling actions (cf. Requirement 1). By means of the graph pattern matching and transformation rules, a multitude of inconsistencies can be diagnosed and handled (Requirements 2 and 3) as well as prioritized (Requirement 4). However, whether the approach is suitable to the domain of automated production systems, in which a multitude of heterogeneous models are being used, needs to be evaluated (Requirement 5).

Herzig et al. [HP14; HQP14] propose a framework for the diagnosis of inconsistencies, in which graph patterns are used to define inconsistency diagnosis rules. In particular, they make use of Resource Description Framework (RDF) for representing models in a common representational formalism (Requirement 1) and SPARQL Protocol and RDF Query Language (SPARQL) for the purpose of specifying inconsistency diagnosis patterns and of matching them against the models (Requirement 2). Additionally, they support in (semi-)automatically identifying semantic overlaps between models (Requirement 1.3) by means of a probabilistic learning approach [Her15]. Nevertheless, assessing and measuring the impact of a diagnosed inconsistency (Requirement 4) or handling inconsistencies (Requirement 3) goes beyond the scope of their work. Furthermore, additional investigations need to be done in order to verify the applicability of their approach for the domain of automated production systems (Requirement 5).

A approach based on Semantic Web Technologies is envisioned by Kovalenko et al. [Kov$^+$14a]. Therein, they aim at an *ontology-based cross-disciplinary defect detecting* concept, in which different ontologies are being used for the purpose of defining the concepts specific to the participating disciplines (Requirement 1). In particular, they exemplify their approach at the hand of a hardware, a control systems as well as a project configuration ontology. Finally, by means of SPARQL graph patterns, inconsistencies are diagnosed within the models (Requirement 2). Whereas by means of the introduced concept, a multitude of different inconsistencies can be *diagnosed*, the authors do not focus on *handling* the respective inconsistencies (Requirement 3). The need for and benefit of implementing a causal analysis of defects is discussed in [Kov+14b]. With their application within an industrial settings (Requirement 5), Kovalenko et al. [Kov+14a] illustrate that Semantic Web Technologies such as RDF and SPARQL are not far from being applied within industrial settings – especially when combining these technologies with Model-Driven Engineering (MDE) approaches or standardized data formats such as AutomationML, their full potentials can be explored [Kov+15]. This is also supported by the fact that database systems, which support the use of Semantic Web Technologies, are more and more available, with appropriate performance and usability characteristics [Ser+13]. Consequently, more and more software prototypes are being developed through applying Semantic Web Technologies; e.g., for the purpose of applying these technologies to AutomationML, an analyser has recently been developed by Sabou et al. [Sab+16]. Furthermore, investigations regarding the applicability of such database systems (e.g., [MSB15; Mor+14]) show that – depending on the particular application scenario – appropriate software architectures exist for integrating heterogeneous models for the purpose of managing inconsistencies in between them.

Wolfenstetter et al. [Wol+18; Wol19] present a model integration framework and software prototype that represents models by means of RDF. Through this common representational formalism, a common knowledge base (Requirement 1) is provided and models can be filtered, e.g., according to the stakeholders' roles that are interacting with the models. In addition, the authors present distinct features that allow for easily importing, manipulating and exporting models. Although with RDF, the basis for inconsistency management is laid, inconsistency diagnosis (Requirement 2) or handling (Requirement 3) are not in scope of their work.

By means of Semantic Web Technologies, in particular OWL and SPARQL, Abele et al. [Abe+13] aim at validating CAEX files regarding pre-defined well-formedness constraints (Requirement 2). Such constraints comprise, e.g., uniqueness of identifiers, compatibility of links as well as compatibility of roles. For the purpose of performing these checks, the authors formulate an OWL ontology that comprises of the necessary semantic elements to be considered by the check (Requirements 1.1 and 1.2). However, their approach is limited to CAEX and has not yet been tested for the heterogeneous models in the automated production systems domain, as well as links in between these models (Requirement 1.3). Furthermore, handling of identified validations of the well-formedness constraints (Requirement 3) is not in focus of Abele et al. [Abe+13].

Similarly, Runde et al. [RF11] make use of the capabilities of OWL for supporting requirements engineering in the building automation systems domain. Therein, OWL ontologies are used together with Semantic Query-Enhanced Web Rule Language (SQWRL) rules to formulate configuration knowledge. By means of this configuration knowledge, room templates can be generated based on the modelled requirements. The combination of CAEX and OWL for a knowledge-based automation systems engineering approach is aimed at by Glawe et al. [Gla+15; GF16]. In particular, the authors extend the expressiveness of OWL by means of Semantic Web Rule Language (SWRL) rules to allow for combining the knowledge modelled in CAEX with security knowledge in an OWL ontology. By means of an interactive visualization concept, the rules are automatically created – hence, comprehensibility (Requirement 5.2) for non-experts in the field of Semantic Web Technologies is increased. However, inconsistency management (Requirements 2 and 3) is not in focus of their work.

Analogously, Ahmad et al. [Ahm+17] make use of pattern matching for the purpose of "[...] ensuring that machine program logic is consistent with process planning requirements". Therein, product, process and resource domain ontologies are used together with a skill model in order to identify, whether logical planning sequences are consistent with a system's program logic. Whereas behavioural inconsistencies in the assembly system domain are in scope of their work, Ahmad et al. [Ahm+17] do not consider a holistic view of the heterogeneous models in the automated production systems domain.

A conceptual approach to inconsistency resolution that makes use of feature diagrams is proposed in [ZV17]. Therein, it is shown that, by means of feature diagrams, "common features of various resolution approaches from different domains [can be generalized]"; however, the authors argue that the "challenging part of the approach lies in the resolution optimization". Hence, an optimization approach to resolving inconsistencies optimally is proposed in [ZLV18]. Therein, genetic algorithms are being applied in order to allow for automated inconsistency resolution. Nevertheless, as of yet the approach is limited to simple inconsistencies (Requirement 5) and does not introduce the means to represent the heterogeneous types of models within a common knowledge base (Requirement 1), which is nevertheless essential for identifying and resolving inconsistencies.

4.2.3. Model Synchronizations

Contrary to logical reasoning and theorem proving approaches, in which a formal system needs to be formulated as the basis to manage inconsistencies, as well as rule- or pattern-based approaches, which make use of a flexible set of rules or patterns, 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 results from executing a synchronization rule. Therefore, 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.

One example for such synchronizational approaches is the concept of Giese et al. [GW06; GW09]. The authors make use of so-called Triple Graph Grammars (TGGs) to specify bidirectional synchronizations between models – hence, these TGGs capture the *links* between model elements formally (Requirement 1.3). Such TGGs contain three essential parts: a *left-side pattern*, a *right-side pattern* and a *correspondence pattern* [Sch94]. By means of respective tags for creating novel model elements, TGG rules can be formulated – therefore, "[a] graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph" [GW06] (that is: if a match can be found for the left-hand side, all objects tagged with a special *create* tag are created). By means of this formalism, a multitude of application scenarios can be identified according to [Sch94] such as *forwards transformation* between a source (left-hand side) and target (right-hand side) model, *backwards transformation* between a target (right-hand side) and source (left-hand side) model as well as *analyses* of the correspondences, e.g., for the purpose of diagnosing inconsistencies. Consequently, with the formalism of Giese et al. [GW06; GW09], a multitude of correspondences can be formulated and, hence, respective rules to diagnose and handle potential inconsistencies between the models can be defined (Requirements 2 and 3). However, whereas transformation rules between individual (software) models are simple to create and maintain, a variety of necessary correspondence rules is expected for the multitude of engineering models in the automated production systems domain (Requirement 1) – consequently, the effort to create and maintain these rules increases with the number of model types that need to be considered. Therefore, although appropriate performance of the presented approach can be expected as incremental model synchronizations are used, the effort to create and maintain these synchronizations for applications in the automated production systems domain must be analyzed and verified (Requirement 5).

To overcome this drawback, Gausemeier et al. [Gau^{+}07] extend the approach presented by Giese et al. [GW06] towards consistency management in heterogeneous models of mechatronic systems. Therein, the authors aim at synchronizing domain-specific models with a domain-spanning so-called *principle solution*, which captures all entities that are relevant to the different involved domains. Their aim is twofold: (1) to detect changes in one domain that are inconsistent to other domains and (2) to propagate changes to the principle solution and to other domains. Consequently, instead of providing $n \times (n-1)/2$ bidirectional synchronizations for n models, the effort is reduced towards n necessary bidirectional synchronizations with the principle solution, leading to a decreased effort in creating and maintaining the transformation rules (Requirement 5). Using the TGG-based formalism, an engineering workflow can be identified that involves four essential application scenarios: (1) forward transformation to create an initial version of a target model as well as the correspondence model from a given source model, (2) backward transformation to identify changes in one model, (3) propagation to update changes from one model to another model and (4) synchronization in order to maintain consistency between the semantically related models. The models that can be addressed by that approach are hierarchical, structural models of the mechatronic systems. Rieke et al. [Rie$^+$12] extend this approach towards managing consistency between behavioral models of the mechatronic system. In particular, they exemplify their concept at the hand of software engineering models (modeled in MechatronicUML) that need to be synchronized with control engineering models (specified in MATLAB/Simulink Stateflow) via the principle solution. Whereas the presented approach allows for diagnosing and handling a multitude of inconsistencies (Requirements 2 and 3) within various models of the mechatronic system (Requirement 1), there is currently a lack in appropriate support of the workflow for managing inconsistencies in heterogeneous engineering models. In particular, the presented concept essentially relies on the modelling approach presented by Gausemeier et al. [Gau$^+$07] – consequently, an integration into existing applications requires tremendous effort for synchronizing engineering models with this principle solution (Requirement 5). Furthermore, what yet needs to be done is to provide appropriate user interaction to support users in identifying critical inconsistencies and appropriate handling actions (Requirement 4).

Kramer et al. [KBL13; Kra17] introduce another approach that makes use of model synchronizations for the purpose of maintaining consistency between models. In addition, in [Kra$^+$15], they introduce the means to propagate changes incrementally between distinct architectural software models. Accordingly, they implement a prototype of their concept using the Vitruvius [KBL13] framework and evaluated it by means of use cases in the software systems domain Requirement 5. Vitruvius has also been applied for the purpose of maintaining consistency within AutomationML models [ABS18] as well as for model variants and versions [Ana$^+$18]. Nevertheless, the heterogeneous models of automated production systems are not in focus these works Requirement 1.

4.2.4. Synopsis

As discussed beforehand, a multitude of different approaches in the field of inconsistency management within heterogeneous engineering models can be found. Most of the approaches originally stem from the software engineering discipline and can be broadly classified into logical reasoning and theorem proving approaches, rule- and pattern based approaches as well as model synchronization approaches. Whereas logical reasoning and theorem proving approaches require a formal system in order to formally derive whether the set of models is consistent or not, rule- and pattern-based as well as synchronization approaches make use of (inconsistency management or synchronization) rules for diagnosing and handling inconsistencies (cf. Requirements 2 and 3). Although being less formal, such rules tend to be more flexible, as rules can be removed and/or added without revising the entire formal system. In terms of extensibility as well as effort to maintain the knowledge base as well as inconsistency management rules (cf. Requirement 5), it can be identified that model synchronizations need a tremendous set of synchronization rules to capture all the correspondences between involved models, whereas describing the situations, in which inconsistencies may occur (e.g., *inconsistency patterns*), provide an appropriate means to efficiently and flexibly specify the inconsistencies.

4.3. Existing Software Tools on Inconsistency Management

Whereas in the preceding paragraphs, mainly related research has been discussed, this section discusses the different available software tools in the field of inconsistency management. For one, *commercial tool suites* that offer different aspects within the context of inconsistency management are discussed (Section 4.3.1). Second, *open-source tools* in the field of MBSE and inconsistency management are described in Section 4.3.2. Section 4.3.3 discusses the main findings.

4.3.1. Commercial Tool Suites

In order for any of these aforementioned approaches to be applicable for industrial settings (Requirement 5), tool support is an essential factor. Within MBSE, most of the commonly known modelling tools is MagicDraw [NoM16]. Therein, MagicDraw supports the means to create and edit models using modelling languages such as UML and SysML. Further on, the means to create custom DSLs is supported – e.g., by means of the profiling mechanisms of UML and SysML. Although support for defining constraints by means of the OCL is provided, managing inconsistencies (Requirements 2 and 3) is not focused on in MagicDraw.

For simplifying the development of industrial automated production systems, a multitude of domain-specific tools is available. For one, the CODESYS Application Composer [3SS16] aims at increasing efficiency during development of IEC 61131-3 applications. However, the overall focus is put on the software development of automated production systems – further discipline-specific engineering models are currently not supported. Integrated, interdisciplinary engineering solutions are, moreover, provided by means of the EPLAN Engineering Configuration tool suite [EPL16] as well as the Siemens COMOS Platform [Sie16]. Although in these tools, it is possible to create interdisciplinary engineering models within a common knowledge base (Requirement 1), thereby establishing the links in between the different, discipline-specific models, inconsistencies are mostly managed either through proprietary rule languages or implicitly. A holistic approach for specifying and diagnosing inconsistencies (Requirement 2) as well as for handling these inconsistencies (Requirement 3) is not in focus of these works. Moreover, although tool providers in the field of Product Lifecycle Management (PLM) such as PTC with its Integrity Modeler [PTC16] more and more start to integrate capabilities to create models by means of, e.g., UML and SysML, a holistic, industrial solution for predefining, configuring and executing inconsistency diagnosis and handling rules (Requirements 2 and 3) within a common knowledge base (Requirement 1) is – to the best knowledge of the author of this dissertation – not yet available.

4.3.2. Open-Source Tools

Especially applied within an academic context, but nevertheless gaining more and more importance for industrial applications, are open-source tool suites. One of the most well-known frameworks in the field of MBSE is the EMF [Ecl16c]. With its extensions – e.g., for the purpose of UML and SysML modelling by means of the Papyrus plug-in [Ecl15] – the EMF supports a multitude of tasks in the fields of MBE and MBSE. Especially as a multitude of standards is supported, such as the Meta Object Facility (MOF) for metamodelling, OCL for constraint specification and evaluation, as well as standards for model-to-model [Obj16] and model-to-text transformations [Obj08], EMF finds its application for manifold use cases. However, also the starting point is made by means of these tools, there is no common concept available for the purpose of inconsistency management (Requirements 2 and 3). One novel framework that extends the EMF for – among others – the purpose of validating models, is the Epsilon Framework [Ecl16b]. Therein, novel languages are provided that allow additional expressiveness compared to OCL – therefore, the formulation and evaluation of inter-model constraints is allowed (Requirements 2 and 3). However, although this framework provides an essential basis and has already been tested for exemplary models of automated production systems in prior work [Fel+16c], an interdisciplinary model knowledge base for engineering models of the automated production systems domain has not yet been developed (Requirement 1).

AutoFocus [for16] puts an essential focus for developing and verifying embedded software systems. Besides requirements specification and analysis, AutoFocus puts emphasis on modelling and simulation, formal verification, design space exploration, code generation as well as testing of software-intensive systems. Although extensions towards additional aspects from the automated production systems domain have been created [Leg+14] with a special focus on interface behaviour modelling, AutoFocus does not focus on the means to manage inter-model inconsistencies among the multitude of models that are involved during engineering of automated production systems (Requirement 1).

4.3.3. Synopsis

The presented software tools address various aspects of the requirements introduced in Chapter 3 – e.g., for the purpose of modelling the multitude of discipline-specific engineering models (Requirement 1). Especially proprietary, commercial tool suites exist, that aim at improving efficiency during engineering of automated production systems. However, a tool suite that incorporates the different engineering models in the automated production systems domain and that allows for defining, diagnosing and resolving inconsistencies (Requirements 2 to 4) is not yet available.

4.4. Summary

Summarizing the related research work, a multitude of different approaches can be found – both in the fields of MBE and MBSE as well as in the field of inconsistency management. Whereas approaches that focus an integrated, model-based systems engineering envision integrated and/or linked models in order to provide a consistent knowledge base (Requirement 1), concepts in the field of inconsistency management put emphasis on the technical perspective of inconsistency management, thereby providing the means to specify inconsistency diagnosis and handling rules (Requirements 2 and 3) and assessing the impact of inconsistencies (Requirement 4). Moreover, manifold tool support can be found, that provides extensible and scalable means and, hence, the basis to develop frameworks for inconsistency management (Requirement 5) – both in industry and academia. However, to the best knowledge of the author of this dissertation, an appropriate approach that allows for managing inconsistencies in the heterogeneous models of automated production systems is not yet available.

Hence, in order to fill this research gap, an appropriate model knowledge base (Requirement 1) must be found, which incorporates the various disciplines that are involved during the engineering of automated production systems. By means of this knowledge base, a framework for managing inconsistencies within and among these models can be provided (Requirements 2 and 3), which supports engineers in identifying and handling potentially occurring inconsistencies (Requirement 4). However, when developing such a framework, the typical constraints within the automated production systems domain – among others, the complexity of these systems as well as the knowledge of experts in this domain – must be considered (Requirement 5).

Chapter 5.

Concept: Diagnosis and Handling of Inconsistencies

This chapter introduces the concept for an approach to diagnose and handle inconsistencies in the automated production systems domain. The concept follows the suggestions in [NER00] as discussed in Chapter 2 and builds on the requirements and assumptions derived in Chapter 3. An architectural overview of the concept is given in Figure 5.1.

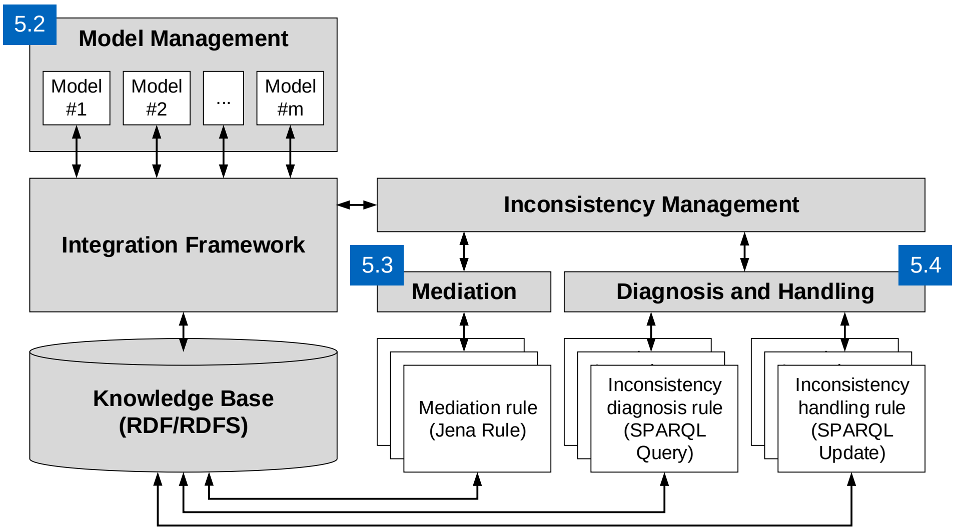

5.2
Model Management
Model #1
Model #2
...
Model #m
Integration Framework
Inconsistency Management
5.3
Mediation
Diagnosis and Handling
5.4
Knowledge Base (RDF/RDFS)
Mediation rule (Jena Rule)
Inconsistency diagnosis rule (SPARQL Query)
Inconsistency handling rule (SPARQL Update)

Figure 5.1. Architectural overview of the concept for diagnosing and resolving inconsistencies in heterogeneous models of the automated production systems domain [extended from Fel+15b]

As discussed in Chapter 2, a fully automatic inconsistency management approach for the automated production systems domain is neither possible nor desirable, as engineers make and revise their decisions throughout the entire system life cycle. As a consequence, a concept for inconsistency management in the automated production systems domain must incorporate the different stakeholders that are involved during the engineering of the system under consideration. Consequently, a stakeholder-centred overview of the inconsistency diagnosis and handling approach is given in Section 5.1, which describes the different parts of the concepts as well as their relations to the stakeholders involved during engineering. The *model management* part of the inconsistency management approach is introduced in Section 5.2, therein describing the different knowledge representations for the disparate, heterogeneous models. Besides discipline-specific engineering models, in many cases, further background information such as knowledge on dependencies between physical units, etc., is necessary in order to draw conclusions on whether an inconsistency occurs or not. As appropriate abstraction techniques are necessary to not only ensure a common syntax, but also common semantics to be used for inconsistency management, respective *mediation* techniques are necessary to mediate between the disparate models (cf. Section 5.3). Based on these components, the means to *diagnose and handle* inconsistencies (cf. Section 5.4) are introduced.

All these components of the concept have been selected based on the requirements introduced in Chapter 3.

5.1. Stakeholder-centred Overview of the Concept

An overview of the concept for diagnosing and handling inconsistencies in between heterogeneous models of an automated production system is, from a stakeholder's point of view, illustrated in Figure 5.2. Therein, distinct *engineering models* are created – such as component lists from a mechanical engineering perspective, electric circuit diagrams in the field of electrical engineering as well as control software. For creating and maintaining each of these engineering models, *discipline-specific engineers* are responsible. At the same time, background knowledge exists – which is, up to now, rarely considered explicitly in current engineering processes. A *knowledge engineer* is envisioned to be responsible for creating and maintaining these *background engineering models*, which comprise of, e.g., unit conversions, device taxonomies, etc. Certainly, all these models are persisted in a domain- or tool-specific format, thereby using *language-specific concepts* – namely, a syntax and semantics that are common to the respective language being used to capture the models. For instance, whereas in mechanical engineering, it is focused on the physical structure of the entire system, electrical engineers focus on a device-oriented structure and software engineers define the functional architecture of the system under investigation [FFV12].

As described in Chapter 3, for the means to manage inconsistencies within and between these models, it is essential to provide a common knowledge base (Requirement 1), which captures the different models within a common syntax and semantics. Therefore, instead of working directly on the language-specific modelling constructs, so-called *virtual models* are used, which are created out of the persisted models by means of model transformations. These virtual models allow (1) to capture the modelled entities and relations within a common syntax (Requirement 1.1), (2) to define common semantics (Requirement 1.2) for the purpose of inconsistency management and (3) to establish the links in between the respective models (Requirement 1.3). Consequently, instead of applying the language-specific concepts being used within the models, *graphs* are used that are generated from the respective models. As the common representational, graph-based formalism for these graphs, the Resource Description Framework (RDF) is used – hence, the starting point for the inconsistency management framework is a graph-based representation of all involved models.

In order to define the structure and concepts to be used for this graph-based model representation, it is essential to define the *discipline-specific concepts* that are captured in the *discipline-specific graphs* – e.g., the mechanics graph, the electrics graph and the software graph. As the basis to specify these concepts, RDF Schema (RDFS) vocabularies are used. These vocabularies specify the entities to be used within the different disciplines – e.g., *Program Organization Units (POUs)* within the software graph or *parts* within the mechanics graph. Consequently, the entities contained in the graph are specified by means of the vocabularies and, therefore, any inconsistency (diagnosis or handling) rule can be defined by means of these vocabularies.

However, defining the individual inconsistencies that may occur between the models can be time-consuming and error-prone: For each discipline to be considered within the inconsistency management framework, respective (possible) inconsistencies in relation to the other disciplines must be specified. Hence, in order to align the diverse, discipline-specific and heterogeneous models and, by that, to allow for defining inconsistency (diagnosis or handling) rules on a more abstract level, a mediation mechanism is used to synchronize these graphs with *domain-specific concepts* – i.e., concepts that are shared by the domain of automated production systems within a *mechatronics graph*. By that, the concepts that overlap in the respective mechatronic disciplines (i.e., mechanical

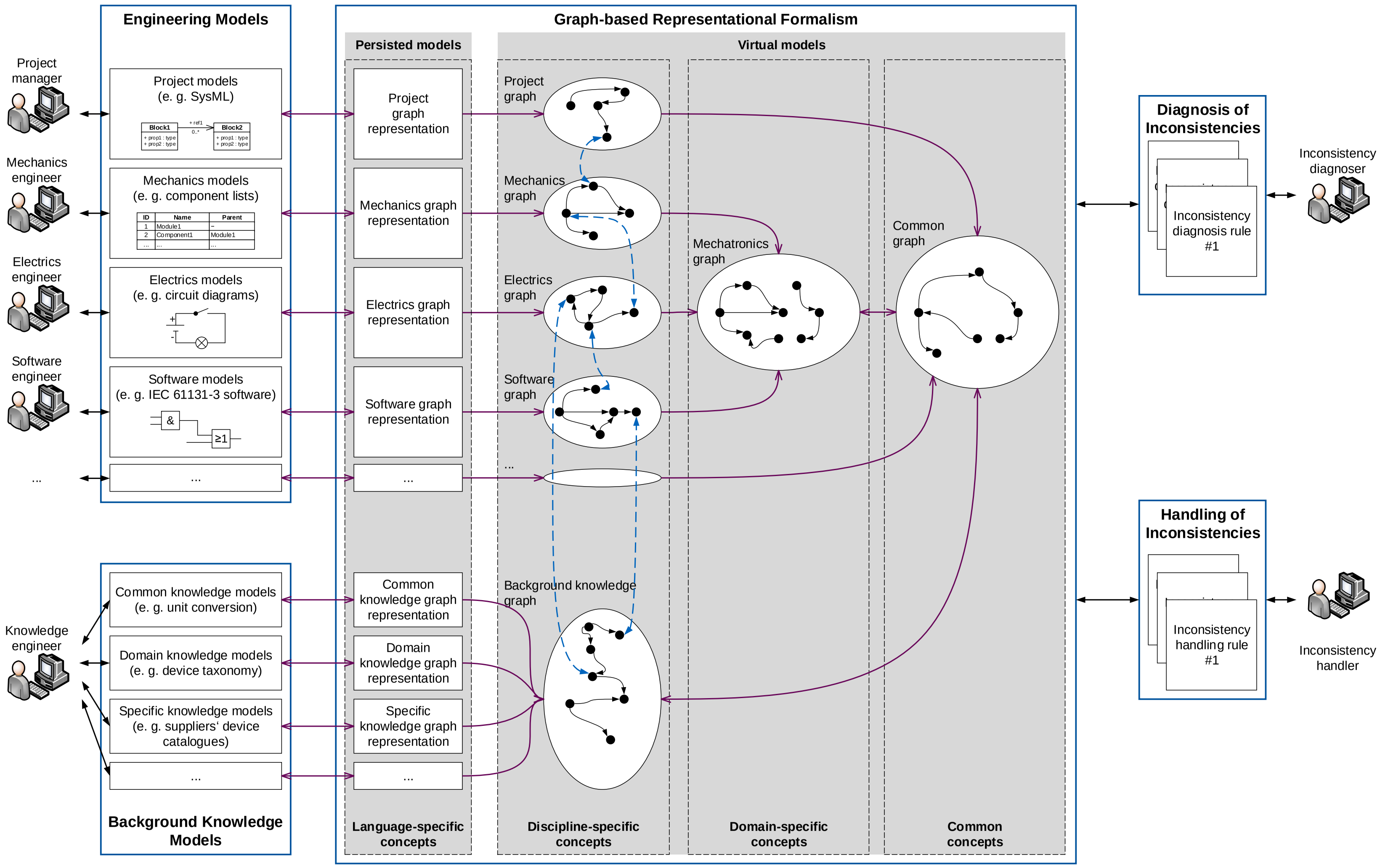

Engineering Models
Project manager
Project models (e. g. SysML)
Block1
+ prop1 : type
+ prop2 : type
+ ref1
0..*
Block2
Mechanics engineer
Mechanics models (e. g. component lists)
ID
Name
Parent
1
Module1
-
2
Component1
Module1
Electrics engineer
Electrics models (e. g. circuit diagrams)
Software engineer
Software models (e. g. IEC 61131-3 software)
&
≥1
...
Knowledge engineer
Common knowledge models (e. g. unit conversion)
Domain knowledge models (e. g. device taxonomy)
Specific knowledge models (e. g. suppliers' device catalogues)
Background Knowledge Models
Graph-based Representational Formalism
Persisted models
Virtual models
Project graph representation
Mechanics graph representation
Electrics graph representation
Software graph representation
Common knowledge graph representation
Domain knowledge graph representation
Specific knowledge graph representation
Project graph
Mechanics graph
Electrics graph
Software graph
Background knowledge graph
Mechatronics graph
Common graph
Language-specific concepts
Discipline-specific concepts
Domain-specific concepts
Common concepts
Diagnosis of Inconsistencies
Inconsistency diagnosis rule #1
Inconsistency diagnoser
Handling of Inconsistencies
Inconsistency handling rule #1
Inconsistency handler

Figure 5.2. Stakeholder-centred overview of the concept for diagnosing and handling inconsistencies in heterogeneous models in the automated production systems domain

engineering, electrical engineering and software engineering) are captured in a mechatronics vocabulary. Moreover, some concepts such as attributes, classes, etc., are common to all involved models. Aligning the graphs with *common concepts* allows to capture a so-called *common graph*. By means of this mediation mechanism, inconsistencies can be diagnosed and handled in an efficient manner.

The *diagnosis* (Requirement 2) and *handling* (Requirement 3) of inconsistencies is realized by means of a rule-based mechanism that accesses, interprets and manipulates the respective graphs. In order to specify these rules, the SPARQL Protocol and RDF Query Language (SPARQL) standard is applied – in particular the query language for the means to specify inconsistency diagnosis rules and the update language to define inconsistency handling rules. From a stakeholder's perspective, individual people are intended to create, maintain and execute these rules – e.g., within the roles of *inconsistency diagnosers* and *inconsistency handlers*. These stakeholders are, accordingly, responsible for diagnosing the inconsistencies, for identifying possible handling actions and for measuring and assessing the respective impact of an inconsistency as well as its handling action (Requirement 4).

As the concept builds upon well-established standards and technologies – namely the RDF and SPARQL – its applicability is not limited to a single domain such as the domain of automated production systems, but can also be applied to further domains or extended towards company- or project-specific purposes (Requirement 5).

5.2. Model Management: Representation of Models

This section introduces the representational formalism to be used within the inconsistency management framework. First, the design rationale for introducing a representational formalism for inconsistency management is discussed in Section 5.2.1. Subsequently, the different vocabularies for representing the engineering models (Section 5.2.2), the background knowledge models (Section 5.2.3) as well as the links in between these models (Section 5.2.4) are introduced.

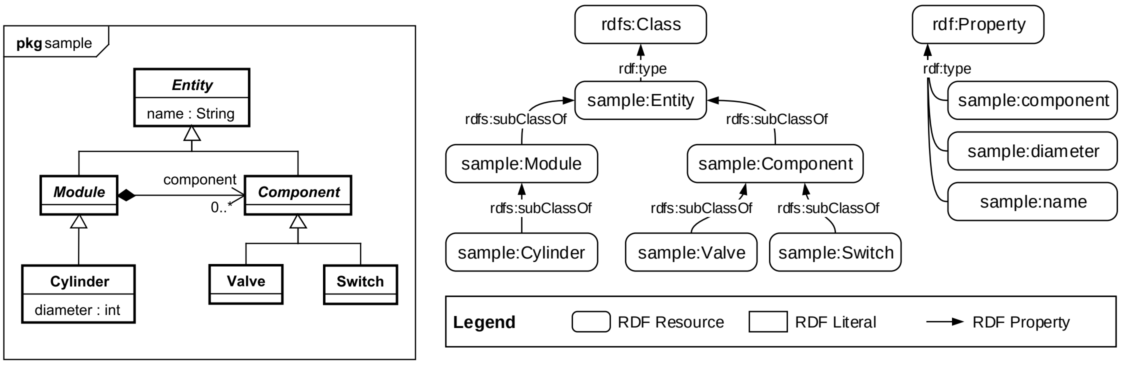
5.2.1. RDF as the representational formalism for models

As discussed in Chapter 3, an essential basis for inconsistency management is the use of a knowledge base, which allows to capture the respective, heterogeneous models to be considered within a common syntax (Requirement 1.1). Moreover, it is essential to provide the means to specify the concepts that are similar to various of the models within common semantics (Requirement 1.2). Finally, the links between the different model instances must be captured (Requirement 1.3). Consequently, an appropriate formalism is required, that allows to transform existing engineering models into a common, representational formalism.

One way to achieve this is the RDF introduced in Chapter 2. The original intention of RDF – to improve the exchange of data on the Web without losing their original meaning [HKR10] – is similar to the challenge of heterogeneous models: Different, heterogeneous data must be represented in a common formalism. Consequently, RDF is envisioned to be used as the formalism to represent the models to be considered for inconsistency management within a central knowledge base.

However, to allow for defining inconsistency diagnosis and handling rules, it is essential to specify the concepts to be used within the RDF graph – that is, to specify the structure of the respective graph. As a consequence, for each model (or discipline) to be considered within the inconsistency management framework, a respective *vocabulary* is needed (see Chapter 2). This vocabulary specifies the entities and relations to be used within the respective graph – hence, the structure of the graph to be created is pre-defined. Consequently, this vocabulary is, with regard to a specific graph that uses this vocabulary, similar to a metamodel for a specific model instance. These discipline-specific vocabularies provide the starting point of the inconsistency management framework – that is, each modelling language or tool to be integrated into such an inconsistency management framework must support the transformation of modelled data into a graph that uses the pre-defined vocabulary. However, instead of capturing all entities within the respective disciplines, it is focused on the ones that are considered as important for inconsistency management – hence, it is not aimed at defining a *world model*, which captures all possible entities of the disciplines in the automated production systems domain.

An example of such a vocabulary is illustrated in Figure 5.3. Therein, within a Unified Modeling Language (UML) class diagram (Figure 5.3a), the concepts *Module* and *Component* inherit from the base concept *Entity*, which specifies a *name* property to be used for all instances. Moreover, *Modules* can contain *Components* via the respective compositional relation *component*. Finally, the Module *Cylinder* with its property *diameter* as well as the Components *Valve* and *Switch* are specified. By means of this vocabulary, the concepts to be used by each specific model instance are defined. The RDF representation of this graph is illustrated in Figure 5.3b. As can be seen, the concepts within the UML class diagram are represented as RDFS classes – the respective properties are defined as RDF properties. For the purpose of clarity and simplicity, the Unique Name Assumption (UNA) is to be used for the vocabularies within the framework – that is, each entity and property being defined have a unique name.



(a) UML class diagram

(b) RDF graph

Figure 5.3. Sample vocabulary represented as a UML class diagram and as an RDF graph

For the purpose of clarity and simplicity, UML class diagrams are used to illustrate the different vocabularies in the following.

5.2.2. Engineering Model Representation

Within the following, the respective vocabularies for capturing the entities within the different engineering models are introduced.

Project vocabulary: Specifying the project's structure

The *project vocabulary* serves as the basis to capture the information that is related to an engineering project – similarly to the resource model presented in [MSD10]. Therein, the customer- and supplier-related data is captured and the bill of material is formed in order to identify the material that is necessary for the engineering project. Consequently, typical sources for the information captured within the specific project graph can be extracted from, e.g., Enterprise Resource Planning (ERP) systems.

An overview of the project vocabulary is given in Figure 5.4. Therein, the *ProjectModel* represents the entry point into the vocabulary – and is the root element of any instance of the project vocabulary. Within the ProjectModel, different *CompanyIndices* can be specified – e.g., in order to capture any information on relevant customers, suppliers or similar stakeholders. Each CompanyIndex consists of a set of *Companies*, which are used to specify, e.g., the *customer* of a *Project* or the *manufacturer* of a specific *Material*. Accordingly, a *MaterialLibrary* is used to capture all available *Material* that can be used within the *BillOfMaterial*.

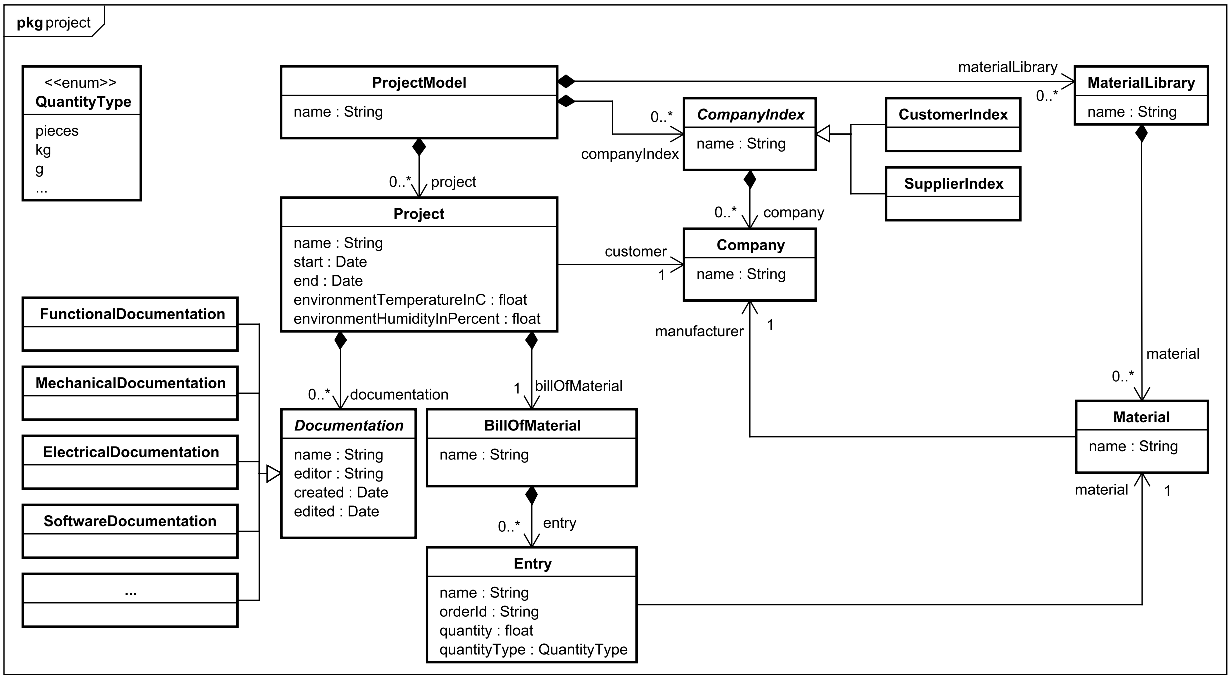


Figure 5.4. Overview of the project vocabulary

Each specific *Project* within the ProjectModel consequently consists of two essential parts: First, the *BillOfMaterial* specifies *Entries*, in which the *Material* to be ordered for the engineered system is captured. These *Entries* – besides the Materials to be used – hold the respective *quantity* as well as the *quantityType* to be ordered for the project. Second, the *Documentation* to be created and maintained for the *Project* is defined – e.g., in the form of a *MechanicalDocumentation*, an *ElectricalDocumentation* or a *SoftwareDocumentation*.

Consequently, with this project-related data, the main information regarding a specific engineering project is collected.

Mechanics vocabulary: Specifying the system's physical layout

Contrary to the *project vocabulary*, which serves as the basis to define an engineering project's structure, the *mechanics vocabulary* aims at defining the physical structure of the system under investigation. Consequently, for the mechanics vocabulary, three essential purposes are envisioned: (1) to specify the parts list from a mechanical perspective, which also defines the composition of the entire system, (2) to define the (material, energy and signal) interfaces between the different mechanical parts and (3) to identify the mechanical functions to be performed by the entire system. Therefore, the information contained within the specific mechanics graph can be extracted from, e.g., typical Computer-Aided Design (CAD) systems. As a basis for defining the interfaces and functions within the *mechanics vocabulary*, the previous investigations by the National Institute of Standards and Technology (NIST) [Hir^{+}02], which include both flow (i.e., interface/connector) definitions and typical functions, were included within the mechanics vocabulary.

Similar to the *project vocabulary*, the *mechanics vocabulary* (see Figure 5.5) involves a root element *PartsModel*, which includes the entities relevant to the specific engineering project. By means of this PartsModel, for one, a *MaterialList* is defined, which includes all *Materials* that are available to the PartsModel.

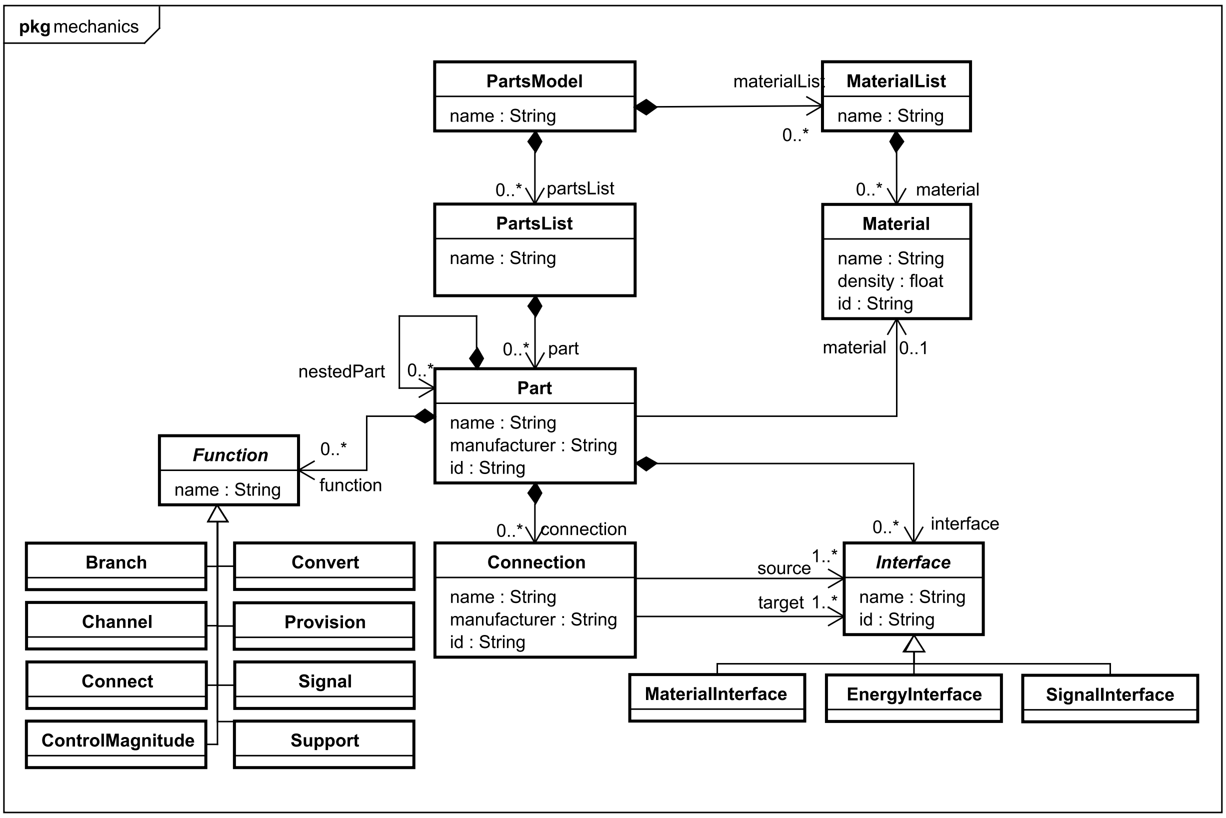


Figure 5.5. Overview of the mechanics vocabulary

To specify the physical composition of the system, a *PartsList* comprises its respective *Parts*. Parts can, in turn, consist of *nestedParts*, which represent their child elements – thereby forming the physical hierarchy of the system. Moreover, Parts can be further specified by means of their *Function* – which can, according to [Hir$^+$02] represent, e.g., a *Branch*, *Convert* or *Channel* Function. Analogously, *Interfaces* can be specified in terms of *MaterialInterfaces*, *EnergyInterfaces* or *SignalInterfaces*. These Interfaces can be further detailed, e.g., by means of the flows defined by Hirtz et al. [Hir$^+$02] – however, these definitions go beyond the scope of this dissertation. Finally, *Connections* can be associated to Parts in order to denote a *Connection* between a set of *source* and *target* interfaces.

By means of the mechanics vocabulary, the information related to the physical decomposition of an automated production system is captured.

Electrics vocabulary: Defining the devices and their connections

Within the *electrics vocabulary*, it is focused on the electrical architecture of the system under investigation. Due to the variety of possible actuators and sensors, also depending on the respective manufacturers of these devices, it is impossible to capture all possible device types in the respective vocabulary. Consequently, it is aimed at a basic framework, which captures (1) devices from an abstract perspective, therefore allowing to unambiguously define different device types and (2) extend the vocabulary according to project- or company-specific purposes in an efficient manner. As a consequence, different sources to acquire the information within the electrics vocabulary are possible, such as the eCl@ss classification [eCl16] as well as respective standards and recommendations such as the Namur NE 100 Recommendation [NAM10]. Hence, in order to provide such a vocabulary for the purpose of inconsistency management, these different sources of information have been aggregated and condensed to capture the information that is necessary for inconsistency management – meaning that this vocabulary makes no claim to completeness and should rather be seen as a first working vocabulary.

It should be noted that the creation, evaluation and continued evolution of the electrics vocabulary were conducted as part of a Semester Thesis at *Technische Universität München* – a detailed documentation of the electrics vocabulary is published in [Fis16].

Overview of the electrics vocabulary. As this vocabulary aims at capturing information from typical electric circuit diagrams, the root element of the electrics vocabulary is the *ElectricCircuit* (see Figure 5.6). The circuit diagram consists of various electric elements – hence, an ElectricCircuit consists of instances of the class *AutomationHardware* – which represents the superclass of all further elements. Essentially, an *AutomationHardware* instance is defined through typical properties such as its *name*, its *id* as well as a (manufacturer-specific) *orderNumber*. Further on, *Interfaces* (e.g., power supply or electrical ports) as well as *Connections* (i.e., wirings from a source to a target interface) can be added to the *AutomationHardware*. Analogously, a *Signal* can be added to an *Interface* in order to specify, what kind of signal is to be expected at the respective interface. The *Signal*'s specification is made by properties such as the respective *voltage*, *current* and *frequency*.

AutomationHardware elements can further be classified into the classes *PLC*, which represents a Programmable Logic Controller (PLC) instance, *Terminal*, which stands for an input or output terminal within the *ControlCabinet*, *Couplers* that represent bus couplers and *Devices*, which represent actual field devices such as *Sensors*, *Actuators* or *SignalAdjustment* devices.

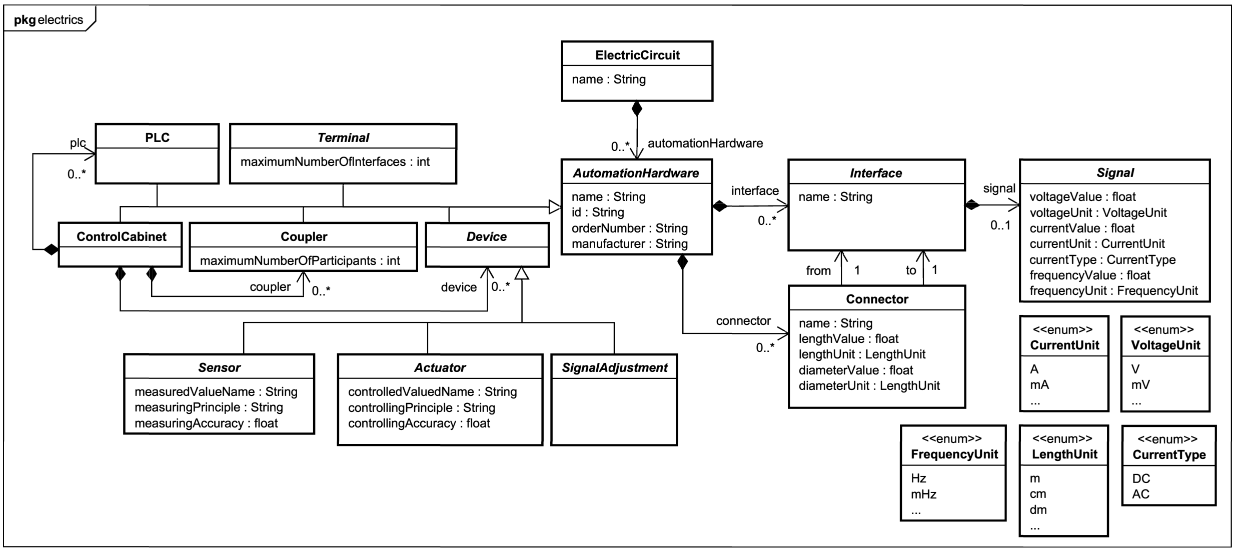


Figure 5.6. Overview of the electrics vocabulary

Devices in the electrics vocabulary. As introduced beforehand, *Devices* can further be classified into *Sensors*, *Actuators*and *SignalAdjustment* devices (see Figure 5.7). For *Sensors* and *Actuators*, the main classifying properties are the respective names of the values that are measured or controlled (see *measuredValueName* and *controlledValueName*), the respective principles that are used for measuring or controlling (see *measuringPrinciple* and *controllingPrinciple*) as well as the accuracy that can be expected for the devices (see *measuringAccurary* and *controllingAccuracy*). Additionally, *Sensors* and *Actuators* can be further classified into binary and analogue devices –

hence, the classes *BinarySensor*, *AnalogueSensor*, *BinaryActuator* and *AnalogueActuator* are introduced as subclasses of *Sensor* and *Actuator*. In order to allow for specification of the values being measured or controlled by analogue devices, the value range can be specified – both for the actual value (see *measuredMinimumValue*, *measuredMaximumValue*, *controlledMinimumValue* and *controlledMaximumValue*) and the transmitted value (see *transmittedMinimumValue* and *transmittedMaximumValue*). Further on, the unit for the measured and controlled values are defined by means of the properties *measuringUnit* and *controllingUnit*. Finally, *conversionFactors* and *offsets* can be specified to denote the conversion between the actual and the transmitted values.

Within the electrics vocabulary illustrated in Figure 5.7, different exemplary *Sensor* and *Actuator* types are included – which can, according to the needs of the respective device types, be extended by further, device-specific properties. Accordingly, exemplary *SignalAdjustment* devices are illustrated, such as *CurrentConverters*, *FrequencyConverters* and *VoltageConverters*.

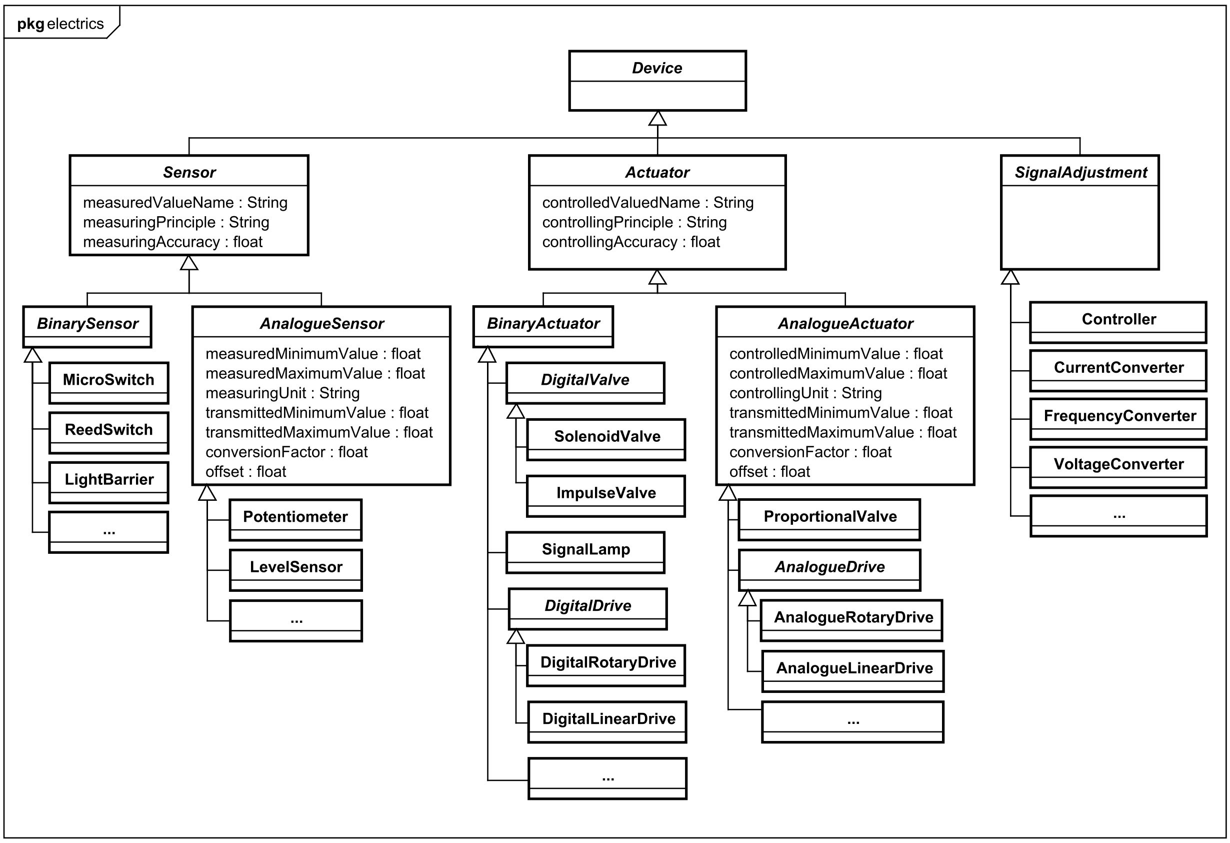


Figure 5.7. Overview of the devices within the electrics vocabulary

Signals and interfaces in the electrics vocabulary. Especially important to the electrics domain, and hence included in the *electrics vocabulary*, are *Interfaces* and *Signals*. For instance, *Terminals* can be broadly classified into *InputTerminals* and *OutputTerminals*. Respective *InputInterfaces* and *OutputInterfaces* can be defined for these Terminals. Moreover, if a bus system is used for the respective device, *BusInterfaces* can be specified – in the excerpt of the electrics vocabulary illustrated in Figure 5.8, some exemplary bus interfaces such as the *EtherCATInterface*, *ProfinetInterface* and *ProfibusInterface* are illustrated. Finally, *DigitalSignal* and *AnalogueSignal* are specified as subclasses of the *Signal* metaclass.

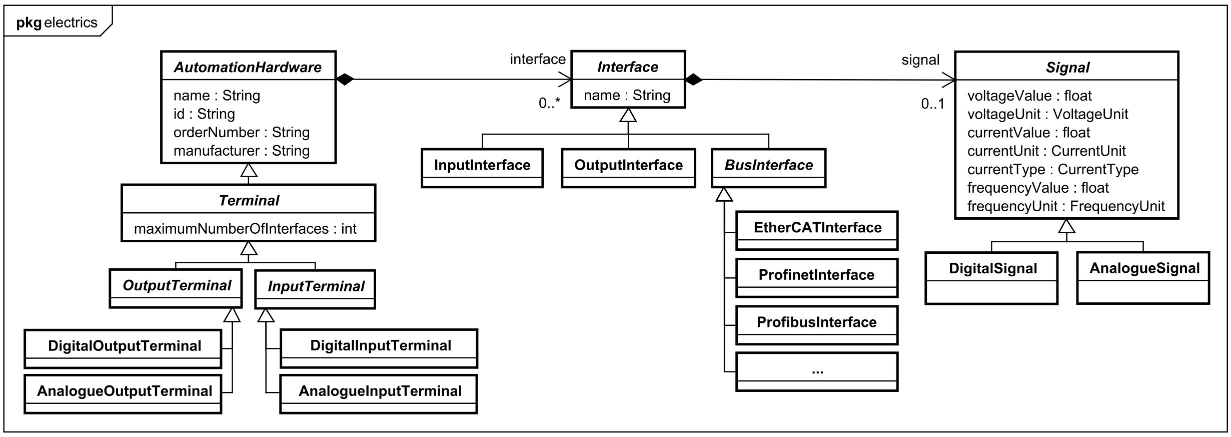


Figure 5.8. Overview of the signals and interfaces within the electrics vocabulary

Software vocabulary: Representing the logical control software architecture

In order to capture the necessary model information from the control software discipline, a *software vocabulary* is introduced in the following. However, in contrast to exchange formats such as PLCopen XML [PLC09b], the software vocabulary aims at capturing the dependencies within IEC 61131-3 software projects. By means of these dependencies, inconsistencies that, e.g., stem from software guidelines can be specified and identified. Hence, in addition to structural information that is captured in exchange formats such as PLCopen XML, the software vocabulary must be enriched by dependencies that are not always available from the (static) project itself, but are generated, e.g., during the compilation process. Among others, these dependencies result from calls between POUs, read or write processes from or to variables, etc. By including this information within the software vocabulary, the necessary data for specifying and diagnosing important inconsistencies, which result from, e.g., company- or project-specific coding conventions or guidelines such as the ones investigated by the PLCopen Promotional Committee 2 [PLC16], can be captured [Fel$^+$16a].

As a consequence, the software vocabulary can be seen as a dependency model that not only captures the structural elements of an IEC 61131-3 project, but also the dependencies between these elements. It is therefore not focused on the actual control code that is implemented within the software, but on the structural entities that form the logical control code architecture. Consequently, this software vocabulary was mainly created according to the IEC 61131-3 standard in its current version [IEC13b], extended by the concepts behind PLCopen [PLC09b] as well as the dependencies, which are necessary for an appropriate structural analysis of the control code [Fel$^+$16a; Fel$^+$16b].

The creation, evaluation and continued evolution of the software vocabulary were conducted as part of a joint research effort together with *Schneider Electric Automation GmbH* and *Technische Universität München* – initial results of the joint work are published in [Fel$^+$16a; Fel$^+$16b].

Overview of the software vocabulary. The root element of the software vocabulary is a *DependencyModel*, which captures both the structural entities of an IEC 61131-3 software project and the dependencies in between these entities (see Figure 5.9). In such *Projects* – which can either represent an IEC 61131-3 *Library* or *Application* – the respective *Configuration* as well as *Types* are captured. In addition, according to the IEC 61131-3 standard [IEC13b], *Namespaces* can be used in order to structure the respective Types. Configurations, in turn, define available *Resources* (e.g., a PLC), on which *Tasks* are defined. Furthermore, *POUInstances* are defined (e.g., a *Program* instance) and assigned to the Tasks that run these instances. According to the IEC 61131-3 standard [IEC13b], the Types that can be defined are either *DataTypes* such as elementary data types or *POUs* such as Programs, Functions and Function Blocks.

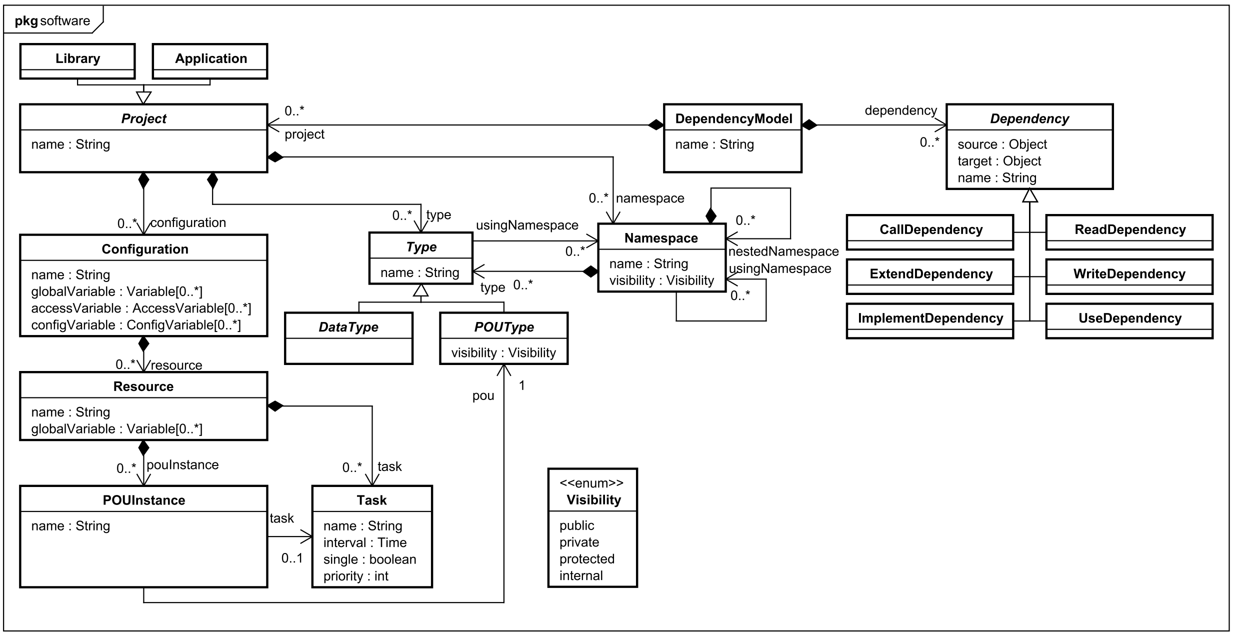


Figure 5.9. Overview of the software vocabulary [enlarged from Fel+16a]

As discussed beforehand, besides these structural entities, (implicit and explicit) dependencies between these entities exist. Such (explicit) dependencies can, for one, result from the relationships and properties that occur within the control software architecture (e.g., a Function Block that *extends* another Function Block). Besides, (implicit) dependencies result from the actual implementation of the respective entities (e.g., a Function Block that *calls* another Function Block). These dependencies are represented by a *Dependency* within the software vocabulary. In addition to *CallDependencies* (i.e., a POU *calls* another POU) as well as *Read-* or *WriteDependencies* (i.e., a Variable is *read* or *written* by a POU), further dependencies exist such as *ExtendDependencies* (e.g., a Function Block *extends* another Function Block), *ImplementDependencies* (e.g., a Function Block *implements* an Interface) as well as *UseDependencies* (e.g., a Type *uses* a Namespace). As the *Dependency* element can point to any object, the *Dependencies* can be extended and used for any relation between elements of the software vocabulary.

Data types in the software vocabulary. A more detailed overview of the *DataTypes* available in the software vocabulary is provided in Figure 5.10. As can be seen from this Figure, IEC 61131-3 defines three basic DataTypes: *ElementaryTypes* represent a set of (pre-defined) elementary data types – e.g., boolean (*BOOL*), integers (*INT*). *GenericTypes* define additional DataTypes, which can be applied to represent hierarchies of DataTypes – e.g., ANY or ANY_ELEMENTARY. *UserDefinedTypes* aim at defining user-defined DataTypes – e.g., *EnumTypes*, *StructTypes*.

Consequently, respective classes are introduced within the software vocabulary, which represent these different DataTypes and which can be used to define the DataTypes of Variables being used within the software graphs. For instance, in order to represent the base types used by a Variable that is typed by means of a *Generic-* or *ElementaryTypes*, enumerations (*Elementary* and *Generic*) are available in the software vocabulary. As *StructTypes* are defined by a set of Variables to be used for the structure, a compositional relation to the *Variable* class is available. An *EnumType* is defined through a set of *EnumValues*, which allow to pre-define a set of names literals within the enumeration. Types that require the definition of a *Range* – e.g., *SubrangeType* and *ArrayType* – are defined through a lower and an upper value. Finally, the *DerivedType* meta class defines a base type it is derived from.

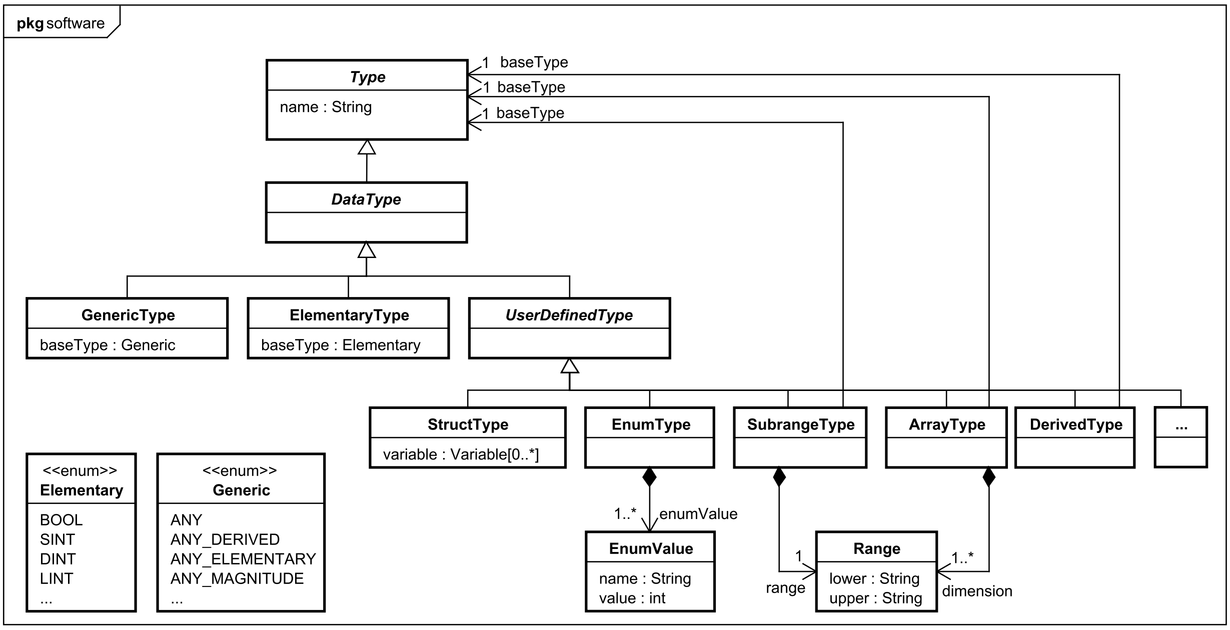


Figure 5.10. Overview of the *DataTypes* within the software vocabulary [enlarged from Fel$^+$16a]

POU types in the software vocabulary. Analogously to the *DataTypes*, a detailed overview of the *POUTypes* that are available within IEC 61131-3 is given in Figure 5.11. Therein, three basic, *ClassicalPOUTypes* are defined by the early standard of IEC 61131-3 [IEC03b]: Functions define non-persistent POUs and yield the calculation of a result value based on given input values. *Programs* define the entry POUs into an IEC 61131-3 Application. *FunctionBlocks* define persistent POUs and calculate output values based on input and persistent internal values.

In order to represent these ClassicalPOUTypes as well as their dependencies, data structures and variables, respective elements are introduced in the software vocabulary. For instance, respective compositional relations to define the Variables within a POU are defined – e.g., to represent input variables (VAR_IN, *inputVariable*), output variables (VAR_OUT, *outputVariable*) or temporal Variables (VAR_TEMP, *temporalVariable*). In addition, as FunctionBlocks, according to the currently available IEC 61131-3 standard [IEC13b] can extend other FunctionBlocks, a respective extension reference (*extendsFunctionBlock*) is available in the software vocabulary.

Analogously to the *ClassicalPOUTypes*, Figure 5.11 introduces the additional, object-oriented *OOPOUTypes* that are specified in the current IEC 61131-3 standard [IEC13b]: *Classes* support the object-oriented paradigm, e.g., through inheritance and interface implementations. *Interfaces* serve the purpose of representing contracts between modular control software units by separating the actual implementation from the specification of a software functionality. *Methods* define operations that can be executed on class instance data.

Similarly to the definition of ClassicalPOUTypes, respective meta classes are specified in the software vocabulary. As the object-oriented extension of IEC 61131-3 also introduces the concept of inheritance, respective properties are available in the vocabulary that specify, e.g., a FunctionBlock or Class that extends another class (*extendsClass*), a Class that implements an Interface (*implementsInterface*) or an Interface that extends another Interface (*extendsInterface*). Abstract classes or methods are denoted through the *abstract* property; if a Method overrides another Method, the *overrides* property is used.

Variables in the software vocabulary. The specification of *Variables* in the software vocabulary is illustrated in Figure 5.12. Within IEC 61131-3, three distinct types of variables are avail-

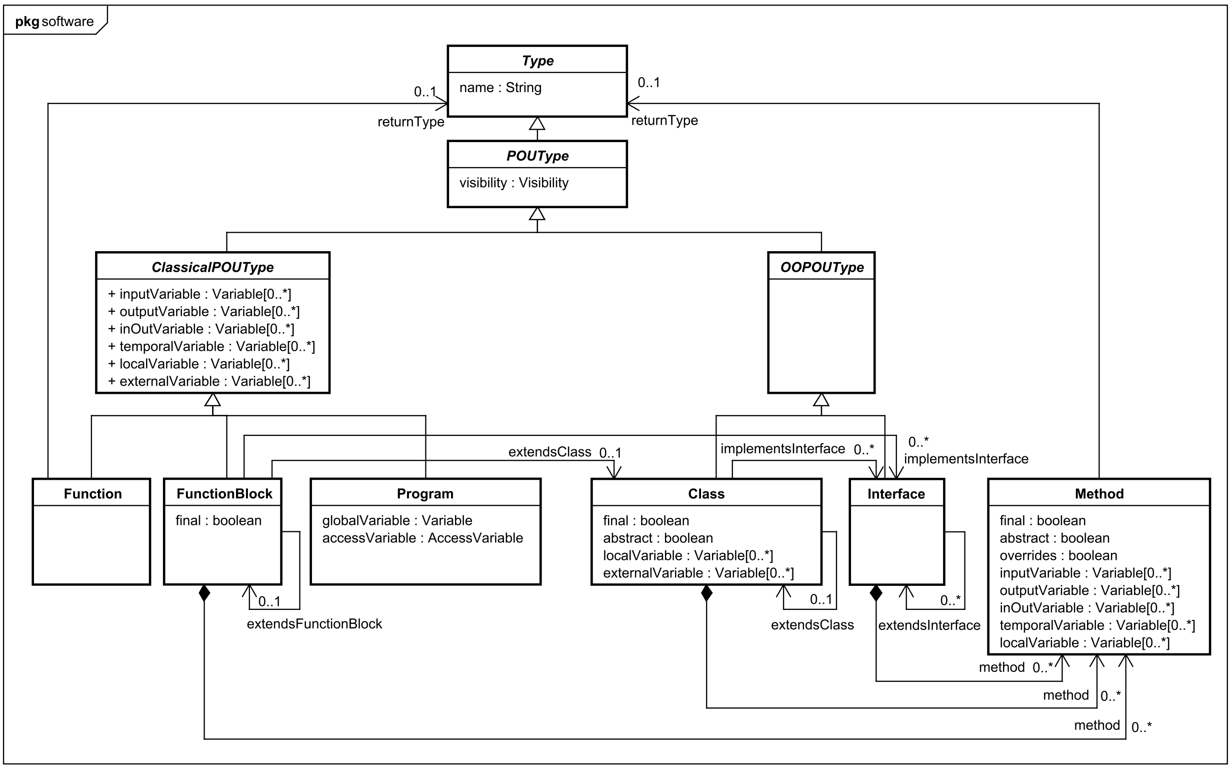

pkg software
Type
name : String
0..1
returnType
0..1
returnType
POUType
visibility : Visibility
ClassicalPOUType
+ inputVariable : Variable[0..*]
+ outputVariable : Variable[0..*]
+ inOutVariable : Variable[0..*]
+ temporalVariable : Variable[0..*]
+ localVariable : Variable[0..*]
+ externalVariable : Variable[0..*]
OOPOUType
extendsClass 0..1
implementsInterface 0..*
0..*
implementsInterface
Function
FunctionBlock
final : boolean
Program
globalVariable : Variable
accessVariable : AccessVariable
Class
final : boolean
abstract : boolean
localVariable : Variable[0..*]
externalVariable : Variable[0..*]
Interface
Method
final : boolean
abstract : boolean
overrides : boolean
inputVariable : Variable[0..*]
outputVariable : Variable[0..*]
inOutVariable : Variable[0..*]
temporalVariable : Variable[0..*]
localVariable : Variable[0..*]
0..1
extendsFunctionBlock
0..1
extendsClass
0..*
extendsInterface
method 0..*
method 0..*
method 0..*

Figure 5.11. Overview of the *POUTypes* within the software vocabulary [enlarged from Fel+16a]

able. For one, *ConfigVariables* (VAR_CONFIG) allow for defining instance-specific location definitions for symbolic variables. These can both be defined within a *Project* or a *Program*; respective instance-specific paths can be defined through the *instancePathAndName* property. *AccessVariables* (VAR_ACCESS) allow for defining communication services, e.g., via remote access to the PLC. Consequently, in addition to their *instancePathAndName* property, a respective *accessType* property is defined for these variables, which can be defined in *Configurations* and *Programs*. Finally, a *Variable* defines variables to be used, e.g., within respective POUs. The *retain* (RETAIN), *non_retain* (NON_RETAIN) and *constant* (CONSTANT) properties can additionally be defined for variables, if used within the control software.

Furthermore, each of the defined variable types can be enriched with *Addresses*, which represent the physical addresses of the respective variables. These Addresses are defined through their *ad-*

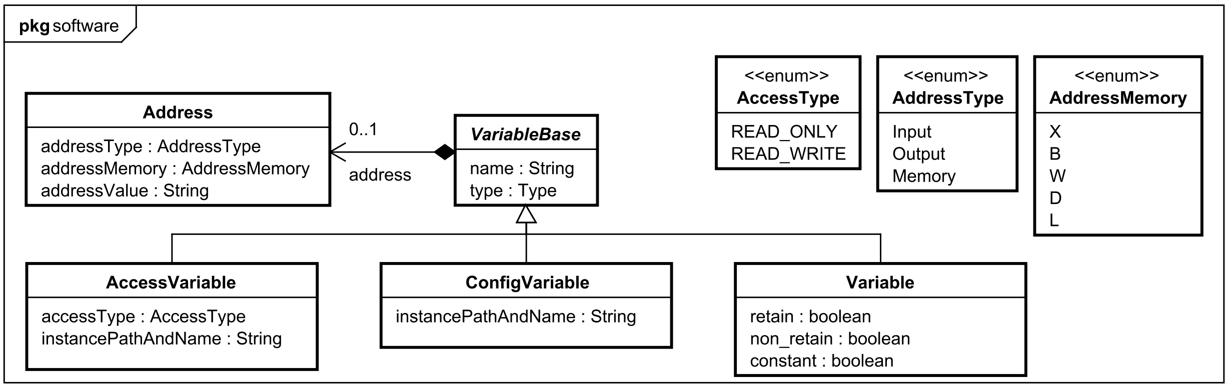


Figure 5.12. Overview of the variables within the software vocabulary [enlarged from Fel+16a]

dressType (*Input* (I), *Output* (Q) or *Memory* (M)), their *addressMemory* (e.g., Single Bit (X) or Byte (B)) as well as through their *addressValue*, which specifies the respective (relative or absolute) address.

5.2.3. Background Knowledge Model Representation

Additionally to the aforementioned vocabularies that are specific to certain disciplines of the engineering within the automated production systems domain, there exists *background knowledge* that is needed for the purpose of inconsistency management. Such background knowledge can involve, e.g., common knowledge such as the knowledge on unit systems, domain knowledge such as device taxonomies as well as specific knowledge such as a supplier's device catalogue. As RDF is used as the formalism to represent models within a common syntax and semantics, in principle any knowledge model can be included within the inconsistency management framework. Hence, by means of additional vocabularies that capture this specific information, such background knowledge can be included for the purpose of inconsistency management.

One typical example is the knowledge on the unit system to be used for inconsistency management. As engineering projects in the automated production systems domain is an interdisciplinary process that is often distributed spatially (e.g., one development team in Europe and another one in Asia), different unit systems are used in a single engineering project. Especially with regard to inconsistency management, it is essential to ensure a consistent use of units – a popular example for the consequences of a lack of inconsistency management with regard to the used unit system is the Mars Climate Orbiter, in which a unit mismatch caused severe monetary consequences [NAS00].

As a basis to provide a standardized means to define such units, the so-called *Quantity, Unit, Dimension and Type (QUDT)* vocabulary is continuously developed and maintained by TopQuadrant and the National Aeronautics and Space Administration (NASA). This collection of ontologies defines "base classes, properties and instances for modelling physical quantities, units of measure, and their dimensions in various measurement systems" [NAS16]. Hence, by means of this vocabulary, it is envisioned to provide an interoperable system that allows for specification and use of such units not only by humans, but also by machines.

Within the context of this dissertation, an excerpt of the QUDT ontologies is used (see Figure 5.13). In particular, for the purposes of this dissertation, it is focused on one-dimensional

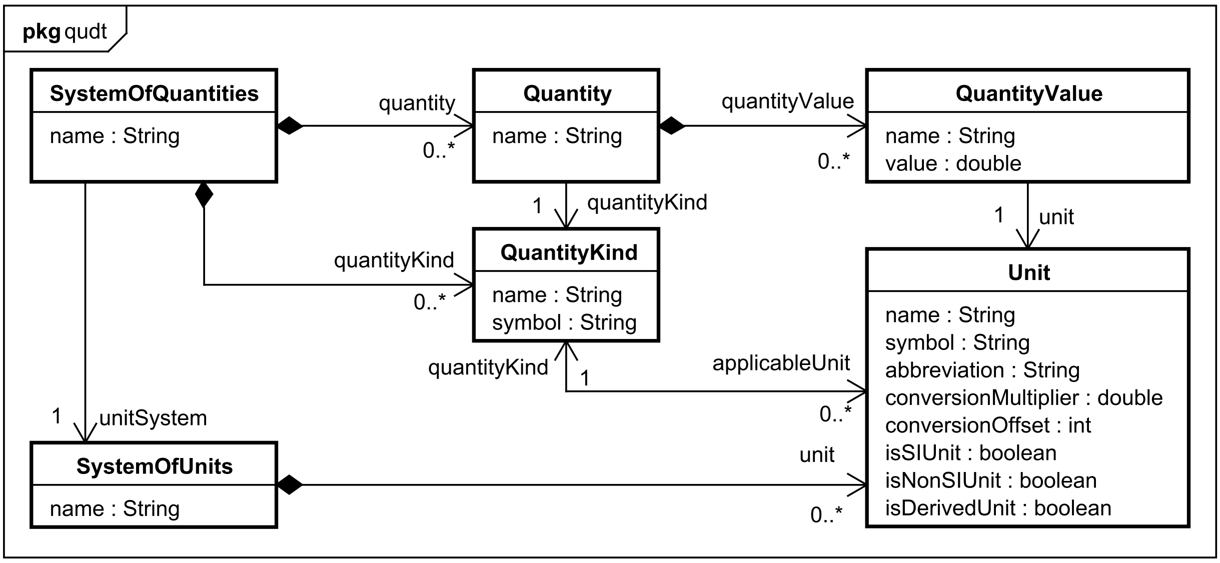


Figure 5.13. Overview of the *QUDT* vocabulary [excerpt according to NAS16])

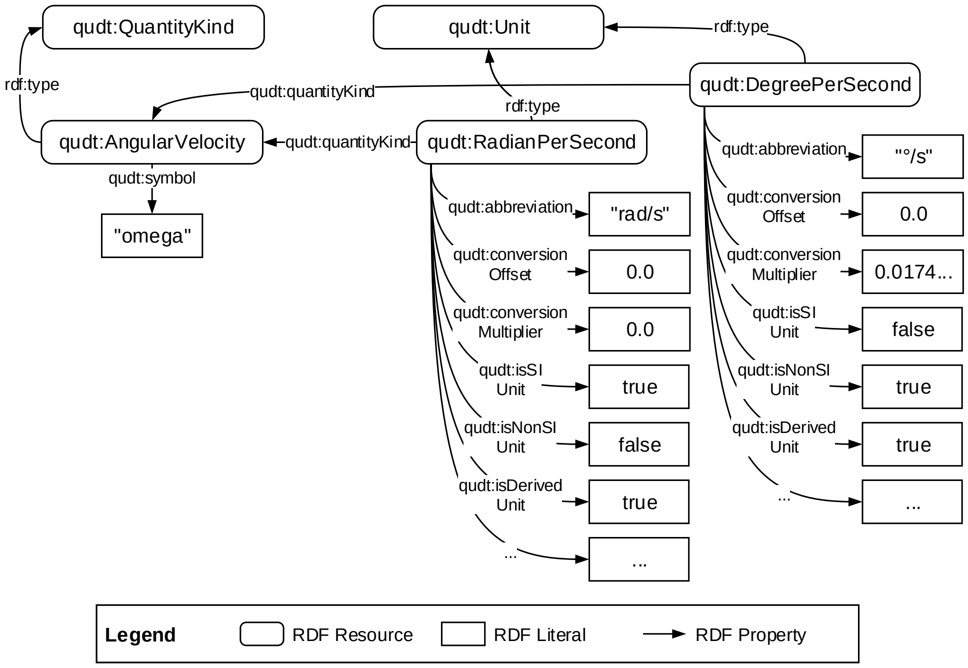


Figure 5.14. Exemplary application of the QUDT vocabulary for the *QuantityKind AngularVelocity* within an RDF graph

quantities and units, therein neglecting unit dimensions that incorporate more than one dimension. QUDT makes use of two main classes: the *SystemofQuantities* class defines the *Quantities* to be used; the *SystemOfUnits* class defines the available *Units*. Consequently, a *QuantityKind* refers to any observable property, which can be quantified in a numerical manner, e.g., length, mass, etc. A *Quantity* is the actual measurement of a *QuantityKind* and, hence, must always be related to a certain thing or value to be measured. A *QuantityValue* therefore represents the certain value measured for a *Quantity*. Accordingly, *Units* represent a unit of measure, which has been chosen as the scale to measure a quantity, e.g., meters, kilograms, etc. Consequently, *Units* are characterised, in addition to to their *symbol* and *abbreviation*, by a *conversionMultiplier* and a *conversionOffset*.

An example applying the QUDT vocabulary for the *QuantityKind AngularVelocity* with its according units *RadianPerSecond* and *DegreePerSecond* is illustrated in Figure 5.14. Given that *RadianPerSecond* is the basic non-SI unit to be used for the *QuantityKind AngularVelocity*, a conversion between *DegreePerSecond* and *RadianPerSecond* can be defined by the *conversionMultiplier* 0.0174... – as illustrated in Figure 5.14. Hence, using the RDF graph in Figure 5.14, the respective relations between the two *Units* can be defined.

As discussed beforehand, further, domain-, company- or project-specific vocabulary can be introduced depending on the envisioned application of the inconsistency management framework – the QUDT is solely a simple example for such background knowledge models.

5.2.4. Linking the Models

Given that a multitude of disparate models is involved during engineering of automated production systems, it is obvious that semantic overlaps exist between the different models (see Chapter 2). As a basis to explicitly capture these semantic overlaps, a *links vocabulary* is introduced.

It should be noted that the creation, evaluation and continued evolution of the links vocabulary were conducted as part of a joint research effort together with *Technische Universität Wien* and *Technische Universität München* – the results were published in [Fel+16c].

An overview of the *links vocabulary* is given in Figure 5.15. This vocabulary describes a *LinkModel*, which consists of multiple *LinkGroups*. Therein *LinkGroups* describe a set of *Links* between two distinct *Models* – a *leftModel* representing the source of a *LinkGroup* and a *rightModel* defining the target of a *LinkGroup*. Consequently, the general *Link* concept is used to describe links between a *leftEntity* (that is: the source of a link) and a *rightEntity* (that is: the target of a link). The general *Link* concept is further refined by means of different link types. By means of these different link types, different semantics for the dependency between two elements can be described:

- An *EquivalentToLink* refers to the equivalence of two model entities. Therein, equivalence is meant to demand the equality of certain properties such as names, identifiers, etc.
- A *DependsOnLink* is used to declare a general (non-specified) dependency between two entities.
- A *SpecializesLink* refers to one element specializing another element. Its opposite link, the *GeneralizesLink*, refers to one element that generalizes another element. This link type is similar to the generalization concept that is used in a multitude of Domain-specific Languages (DSLs).
- A *RefinesLink* aims at specifying a refinement between two entities. Accordingly, its opposite, the *AbstractsLink* refers to the abstraction between two entities. This link type is to be used in a similar manner to the *refines* relation in Systems Modeling Language (SysML) requirement diagrams.
- A *SatisfiesLink*, similar to *satisfy* relations in SysML requirement diagrams, refers to one element satisfying, e.g., a requirement.
- Further, company- or project-specific links can be added to the *links vocabulary*.

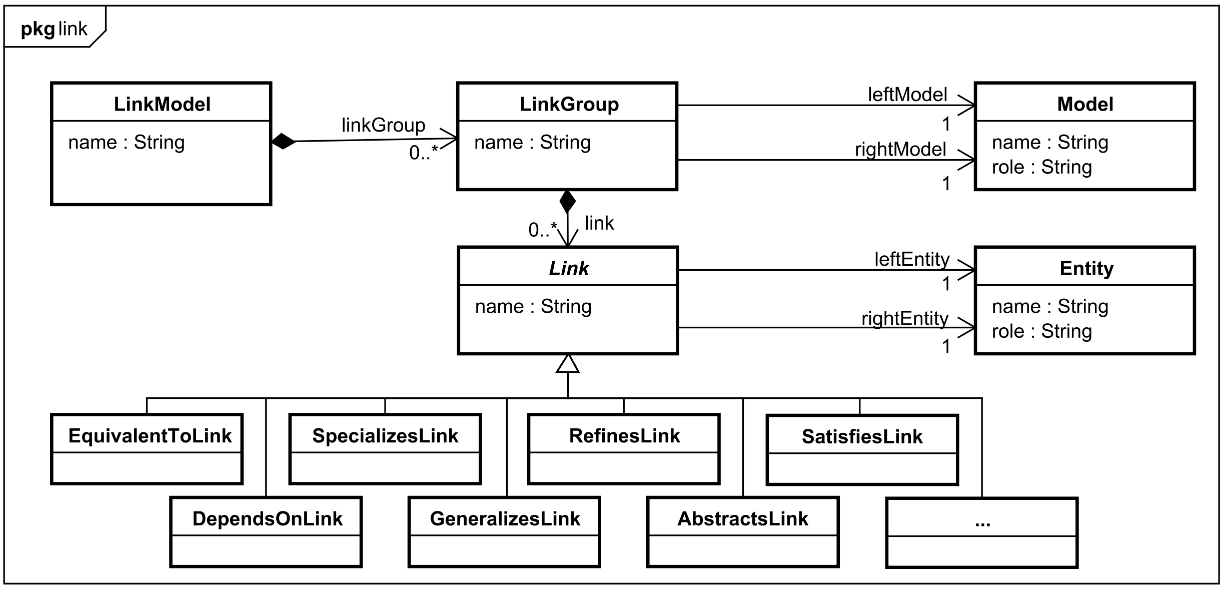


Figure 5.15. Overview of the links vocabulary [extended from Fel$^+$16c]

By means of the *links vocabulary*, a multitude of semantic overlaps between disparate models of the automated production system can be captured. Based on these links, inter-model inconsistencies can be diagnosed and handled.

5.3. Mediation: Mediation Between Heterogeneous Models

Whereas RDF serves as the basis to represent heterogeneous models within a common representational formalism, the multitude of disparate engineering models within the automated production systems domain demands for a mechanism that simplifies the specification, diagnosis and handling of inconsistencies. In particular, it can be argued that some entities modelled in disparate models are semantically overlapping – leading to similar or identical entities and properties being modelled in the different languages, but referring to common semantic concepts (Requirement 1.2).

A comprehensive example is the *naming* attribute: A multitude of applications demand naming conventions (such as medial capitals for compound identifiers – namely, the *UpperCamelCase* style). In order to ensure that such a naming convention is not violated within n disparate models, n distinct inconsistency diagnosis (and according handling) rules need to be specified. This complexity increases when investigating the links between the respective models. An example is the demand for identical names of entities that are linked to each other: If n heterogeneous models were considered for this inconsistency, $2 \cdot n$ inconsistency diagnosis (and according handling) rules would be needed to ensure all possible cases. It is, therefore, inevitable to provide an appropriate abstraction mechanism that allows for unifying the distinct vocabularies – at least to some degree. Especially as similarities exist between the different vocabularies – such as the concept of classes and their respective instances, as well as the existence of properties – common concepts between the different disciplines can be found. By formulating respective mappings between the discipline-specific vocabularies and the more general ones, graphs that use the previously defined vocabularies can be mapped to more abstract vocabularies. These mappings are formulated by means of rules, which are henceforth referred to as *mediation rules*.

Within the following, the design rationale for introducing additional mediating vocabularies and for using rules for mediating between multiple vocabularies is presented (Section 5.3.1). Subsequently, the two mediating vocabularies that are used within the context of this dissertation, namely the mechatronics and the common vocabulary, are introduced (Section 5.3.2), followed by the specification of the rules for mediating between the different vocabularies (Section 5.3.3).

5.3.1. Rules and mediating vocabularies as the basis for effective management of inconsistencies

In order for any inconsistency management framework to be efficient, it is essential to provide a mechanism for abstracting the multitude of involved models into a common "denominator" – that is, to capture the common semantic elements (Requirement 1.2) that are similar to all of the involved modelling languages. It can be argued that – at least at some level of (semantic) abstraction – there exist concepts that are common to a specific domain (such as the automated production systems domain) or even common to all domains. Popular examples for such commonalities are the concepts of *instantiation* or *properties* such as values or names. Consequently, using more abstract vocabularies, inconsistencies among and between multiple, disparate models can be managed.

Clearly, the definition of such common concepts may not always require the full expressiveness; thus, those more abstract vocabularies are semantically much weaker than discipline-specific ones. The concept of *mediating* between disparate vocabularies is illustrated in Figure 5.16. As can be seen from the Figure, for the purpose of inconsistency management in the automated production systems domain, it can be distinguished between three distinct types of vocabularies. For one, *discipline vocabularies* (such as the ones introduced beforehand) capture the information that is relevant to a particular discipline, e.g., mechanics or electrics. These specific vocabularies make use of the full expressiveness of the respective discipline. However, it can be argued that, due to semantically overlapping fields (Chapter 2), concepts can be found that are relevant to an entire domain – hence, resulting into *domain vocabularies*. An example is the mechatronics domain,

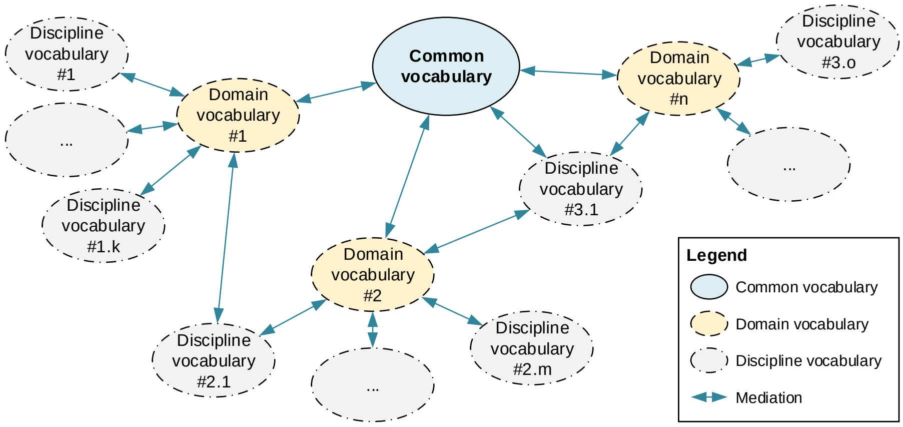


Figure 5.16. Semantic mediation between discipline, domain and common vocabulary [extended from FKV16])

which incorporates, for instance, the concept of a mechatronic *component* or *module*. Such domain vocabularies are semantically weaker than discipline vocabularies, as they do not need the full extent of expressiveness of the involved disciplines. Certainly, overlaps between multiple domains can exist; therefore, there may also exist mediations from one single discipline vocabulary to a set of domain vocabularies. Finally, it is obvious that some concepts are relevant to all domains – this is represented through a *common vocabulary* in Figure 5.16. Such common concepts involve, e.g., the concept of a class-instance relationship (i.e., *instantiation*) or the concept of an object's *property*. Therefore, the expressiveness of such a common vocabulary will be exceptionally weak, as it only includes the concepts that are common to all domains.

The benefits in applying such a mediation concept are manifold. As the more abstract (domain and common) vocabularies represent semantically overlapping concepts, these vocabularies ideally remain unchanged when incorporating additional disciplines or modelling languages within an inconsistency management framework. Hence, said process will only need (1) an appropriate RDF representation of the respective discipline or language and (2) the adequate set of mediation rules for mediating between the novel discipline or language and the dependent vocabularies.

An illustrative example of such a mediation is illustrated in Figure 5.17. Therein, two distinct modelling languages (i.e., *discipline vocabularies*), namely the vocabularies *sampleA* (Figure 5.17a) and *sampleB* (Figure 5.17b), are illustrated together with a sample *common vocabulary* (Figure 5.17c) that should be used as the common "denominator" between these languages – for this illustrative example, a domain vocabulary is omitted. Within the example, the *sampleA* vocabulary aims at defining a *Type Taxonomy*. Therein a hierarchical assignment of *nestedTypes* is made and, with the *massInKg* attribute, further annotated with additional information. Similarly, the *sampleB* vocabulary provides the means to formulate a *Hierachy* of *Entities*, which are either *Modules* or *Components*. Accordingly, *nestedEntities* are defined for *Modules*, thereby forming the hierarchy of the entire system. By means of the *MassProperty*, the mass values of each *Module* or *Component* are defined. Certainly, both vocabularies have semantically overlapping elements and, hence, are mediated with the *sampleCommon* vocabulary. Within this *common vocabulary*, the hierarchies are aligned by means of the *Element* class, which contains nested Elements (property *containsElement*) as well as *Properties*.

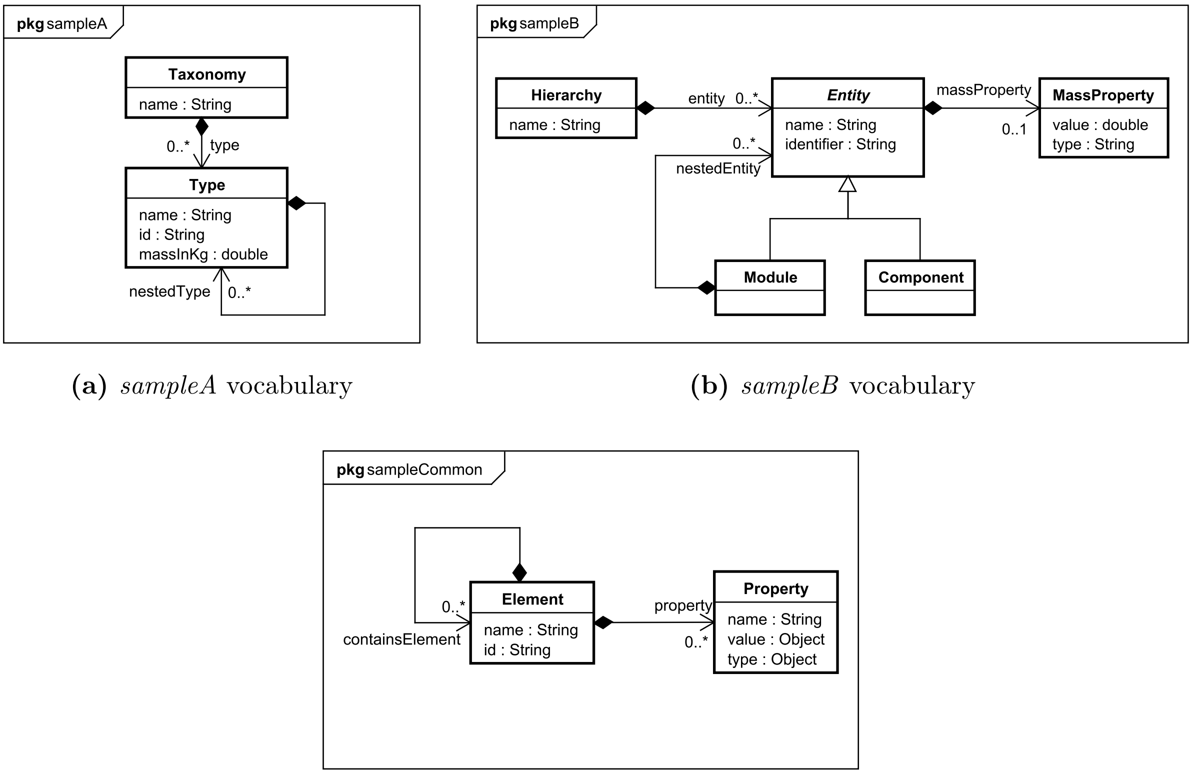
To allow for mediating between the distinct vocabularies, mediation rules are formulated. These mediation rules are similar to graph transformation rules, as they extend the existing graph with additional information according to the given rule. Within the following mediation rules, the vocabularies are denoted by means of indicators: *A* for *sampleA*, *B* for *sampleB* and *C* for *sampleCommon*.

As denoted in Equations (5.1) and (5.2) *Taxonomy* and *Type* instances in the *sampleA* vocabulary are mediated with *Element* instances in the *sampleCommon* vocabulary.

$$Taxonomy_A(x) \rightarrow Element_C(x) \tag{5.1}$$

$$Type_A(x) \rightarrow Element_C(x) \tag{5.2}$$

Accordingly, the *id*, *name*, *type* and *nestedType* properties (Equations (5.3) to (5.6)) are mediated with according properties in the *sampleCommon* domain. For the *mass* property (Equation (5.7)), an according *Property* is created in the *sampleCommon* domain. As mediation rules are unidirectional, traceability between the mediated property as well as the original triple that produced the property must be ensured – hence, the triple is bound to *originalTriple*.



(a) *sampleA* vocabulary

(b) *sampleB* vocabulary

(c) *sampleCommon* vocabulary

Figure 5.17. Exemplary mediation use case: mediation between two sample vocabularies *sampleA* and *sampleB* with a common vocabulary *sampleCommon*

$$
\begin{aligned}
id_A(x,i) &\rightarrow id_C(x,i) && (5.3)\\
name_A(x,n) &\rightarrow name_C(x,n) && (5.4)\\
type_A(x,t) &\rightarrow containsElement_C(x,t) && (5.5)\\
nestedType_A(x,t) &\rightarrow containsElement_C(x,t) && (5.6)\\
massInKg_A(x,v) &\rightarrow Property_C(p) \land property_C(x,p)\\
&\land value_C(p,v)\\
&\land name_C(p,\text{“}Mass\text{”})\\
&\land type_C(p,\text{“}kg\text{”})\\
&\land originalValueTriple_C(p,triple)\\
&\land subject(triple,x)\\
&\land predicate(triple,massInKg_A)\\
&\land object(triple,v) && (5.7)
\end{aligned}
$$

Analogously, mediation rules are formulated for the *sampleB* vocabulary. The main metaclasses in the *sampleB* vocabulary – namely *Hierachy*, *Entity* and *MassProperty* – are mediated to the according metaclasses *Element* and *Property* in the *sampleCommon* vocabulary by means of the mediation rules in Equations (5.8) to (5.10).

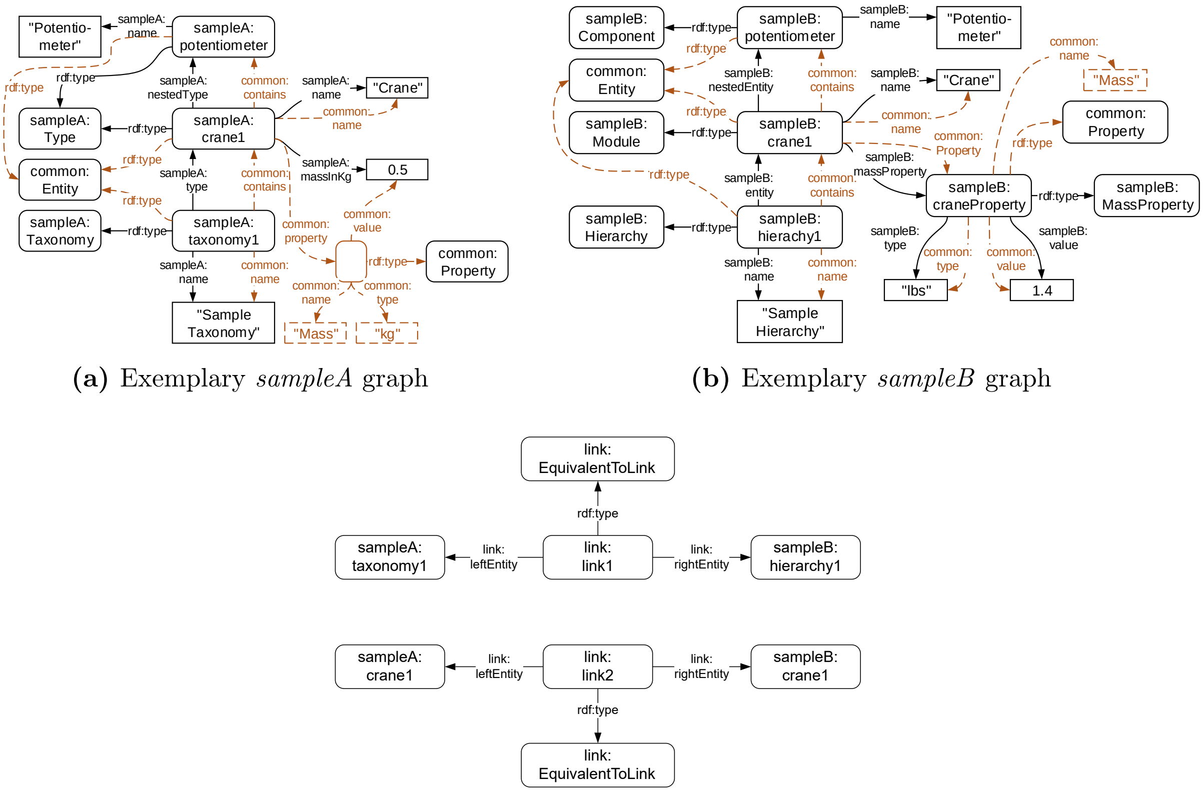
$$
\begin{aligned}
Hierarchy_B(x) &\rightarrow Element_C(x) && (5.8)\\
Entity_B(x) &\rightarrow Element_C(x) && (5.9)\\
MassProperty_B(x) &\rightarrow Property_C(x) && (5.10)
\end{aligned}
$$

Similarly, the rules in Equations (5.11) to (5.14) are formulated to mediate the *identifier*, *name*, *nestedEntity* and *massProperty* properties with according ones in the *sampleCommon* vocabulary. For the *massProperty* (Equation (5.15)), an according *property* relation is generated within the *sampleCommon* vocabulary. Accordingly, the *originalTriple* captures the initial triple that produced the newly created property in the common vocabulary.

$$
\begin{aligned}
identifier_B(x,i) &\rightarrow id_C(x,i) && (5.11)\\
name_B(x,n) &\rightarrow name_C(x,n) && (5.12)\\
entity_B(x,t) &\rightarrow containsElement_C(x,t) && (5.13)\\
nestedEntity_B(x,t) &\rightarrow containsElement_C(x,t) && (5.14)\\
massProperty_B(x,p) \land value_B(p,v) \land type_B(p,t) &\rightarrow Property_C(p) \land property_C(x,p)\\
&\land value_C(p,v)\\
&\land name_C(p,\text{“}Mass\text{”})\\
&\land type_C(p,t)\\
&\land originalValueTriple_C(p,triple)\\
&\land subject(triple,p)\\
&\land predicate(triple,value_B)\\
&\land object(triple,v) && (5.15)
\end{aligned}
$$

An example of the mediation between the *sampleA* and *sampleB* vocabularies as well as the *sampleCommon* vocabulary is illustrated in Figure 5.18. As can be seen from the Figure, both exemplary model graphs represent a *Crane* module with its according *Potentiometer* using the disparate concepts of the *sampleA* and *sampleB* vocabularies. By means of the mediation rules discussed above, the *common* vocabulary is used to describe the different concepts within the graphs using the common concepts *Property* and *Entity* as well as the respective properties. An according link model describes the relations between the two exemplary graphs.

Consequently, by means of the mediation from the two distinct discipline vocabularies with the common vocabulary, involved models can be brought to a common abstraction level, thereby not only using a common syntactical formalism (Requirement 1.1), but also common semantic concepts (Requirement 1.2), that are identical or similar to both disciplines. For the purpose of inconsistency management, the benefits of such a mediation can already be observed for the illustrative example in Figure 5.18: Without the *sampleCommon* vocabulary, an inconsistency (diagnosis or handling) rule that refers to the *mass* values specified in both vocabularies would require at least two distinct formulations – one for the *sampleA* vocabulary and one for the *sampleB* vocabulary. However, after mediating both vocabularies with the *sampleCommon* vocabulary, the necessary formulations are reduced to one single inconsistency (diagnosis or handling) rule, which refers to the *sampleCommon* vocabulary and, hence, is independent from other vocabularies. Consequently, the mediation mechanism is especially helpful when incorporating a set of distinct, heterogeneous (discipline or domain) vocabularies for the involved engineering models – as is the case for the engineering in the automated production systems domain.



(a) Exemplary *sampleA* graph

(b) Exemplary *sampleB* graph

(c) Exemplary *link* graph

Figure 5.18. Mediation between exemplary graphs using the *sampleA* and *sampleB* vocabularies and the *sampleCommon* vocabulary (inferred information is denoted in dashed lines)

5.3.2. Mechatronics vocabulary and common vocabulary: Representing common concepts

This section introduces the two mediating vocabularies used within this dissertation – namely, the vocabulary for the domain of mechatronics as well as a common vocabulary.

Mechatronics vocabulary: Capturing common mechatronic concepts

Within the *mechatronics vocabulary*, the focus lies on mediating between the disciplines of the mechatronic engineering domain, i.e., between the mechanical, electrics and software disciplines. Consequently, it is essential to capture the common concepts from the previously introduced *mechanics*, *electrics* and *software vocabularies.*

As the basis to develop such a mechatronics vocabulary, the prior work of Kernschmidt et al. [Ker$^+$14] is used, which envisions an integrated model-based engineering process for mechatronic production systems. A first draft of a metamodel for such an integrated, mechatronic modelling language was published in [FKV16] – the continuous evaluation and improvement is unpublished up to the date of writing this dissertation, but will be published in the near future in [Ker17]. This metamodel serves as the basis for the mechatronics vocabulary described in the following.

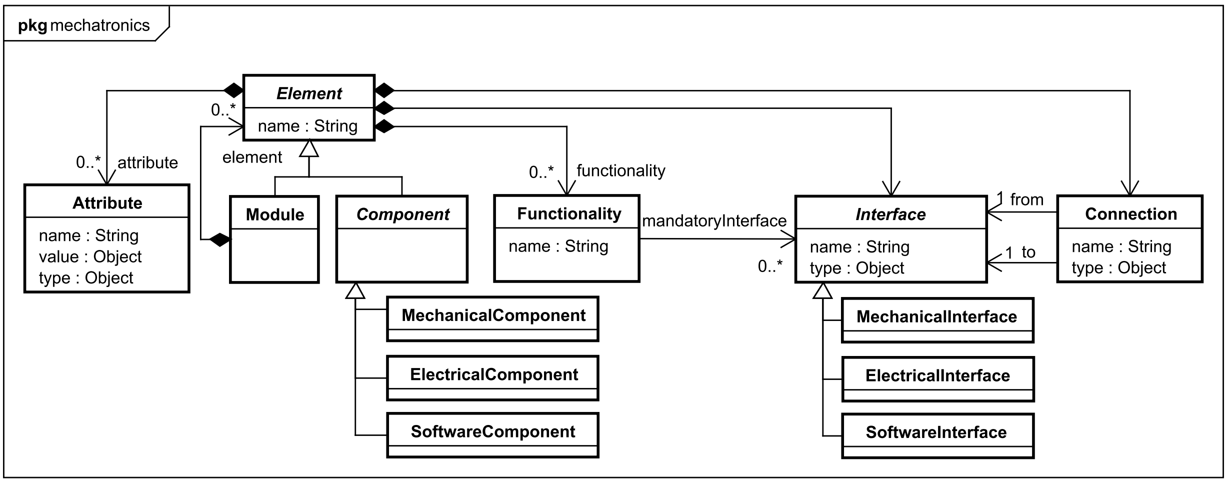


Figure 5.19. Overview of the mechatronics vocabulary [according to FKV16; Ker17]

Within the *mechatronics vocabulary* (see Figure 5.19), mechatronic *Elements* are described – namely either mechatronic *Modules* that can comprise of further *Elements* or discipline-specific *Components.* Such *Components* refer to respective discipline-specific elements and are, hence, further divided into *MechanicalComponents*, *ElectricalComponents* and *SoftwareComponents.* By means of these basic concepts, a distinction between the different discipline-specific entities can be made.

Further on, *Elements* can be described by means of *Attributes*, *Functionality* and *Interfaces.* Therein, *Attributes* refer to a *name – value – type* combination and represent any particular attribute that describes an *Element*, e.g., a mass attribute, a velocity, etc. In turn, *Functionality* refers to the respective function intended to be performed by an *Element.* Respective *Interfaces* define the discipline-specific of a mechatronic *Element*, e.g., a form closure from a mechanical perspective, a bus interface from an electrical viewpoint and a certain operation from a software point of view. Respective *Connections from* and *to* these *Interfaces* represent, e.g., wirings, pipes, etc. Accordingly, the discipline-specific vocabularies can be mediated with the *mechatronics* vocabulary.

Obviously, this mechatronics vocabulary, as discussed beforehand, does not allow for the full expressiveness of the discipline-specific ones. Rather, it aims at capturing the necessary information that is overlapping in the involved discipline vocabularies – namely, the *mechanics*, *electrics* and *software vocabulary.* Accordingly, such a domain vocabulary can be adapted to the specific needs of a company or project – thereby putting emphasis on the concepts that are relevant for inconsistency management.

Common vocabulary: Capturing concepts that are common to all disciplines

Whereas the *mechatronics vocabulary* addresses the concepts from specific disciplines, the *common vocabulary* aims at a generic framework to consider the concept from all disciplines. As a consequence, such a *common vocabulary* comprises the concepts that are common to all involved disciplines – similar to property modelling approaches such as the one proposed by Hadlich and Diedrich [HD13].

The common vocabulary used within this dissertation resulted from a cooperation between the *Georgia Institute of Technology* and different institutes at the *Technische Universität München.* The principal results of this joint research work were published in [Fel$^+$15b; Fel$^+$15a] and have been extended in [Her15; FKV16]. These prior works serve as the basis for the common vocabulary used throughout this dissertation.

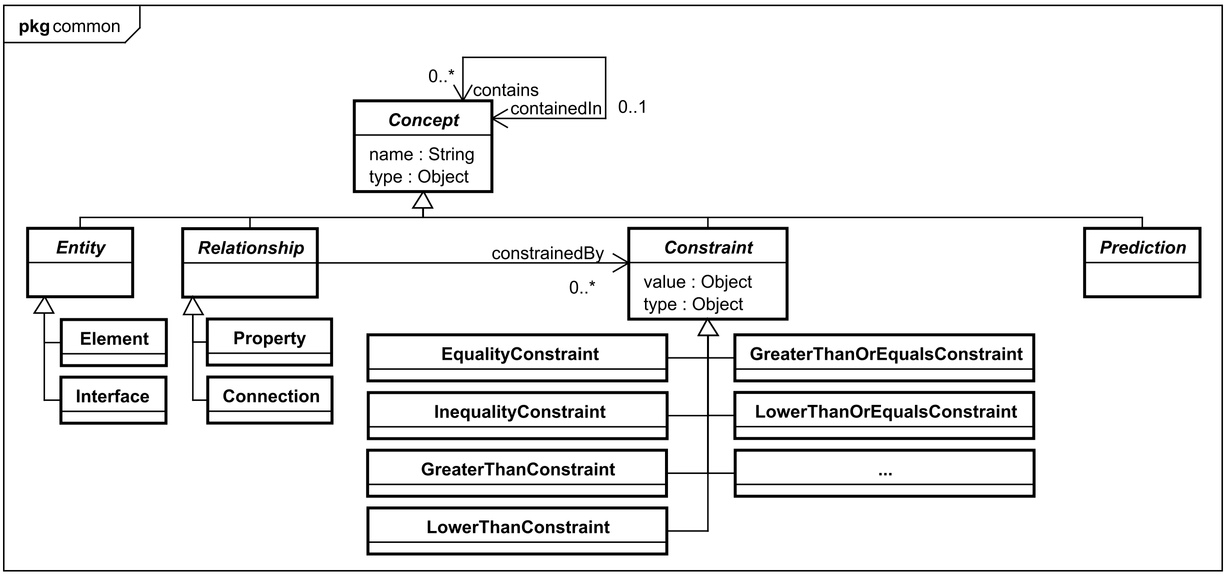


Figure 5.20. Overview of the common vocabulary

As the basis for the *common vocabulary*, the metaclass *Concept* is defined. This metaclass also serves as the root for further classes, which inherit from the *Concept. Concepts* are specified through their *name* and respective *type.* By that, *Concepts* within the common vocabulary can basically represent any existing thing or entity within the involved engineering models.

Concepts are further divided into four main classes: *Entities* – which are either *Elements* (i.e., physical or software elements) or *Interfaces* to the environments. *Relationships* denote either *Properties* that describe *Entities* or *Connections* between *Interfaces. Constraints* represent (mathematical, logical, etc.) conditions that are imposed, e.g., on a *Property.* Among others, typical *Constraints* are *EqualityConstraints* (i.e., *value* must be equal to constrained value), *GreaterThanConstraints* (i.e., *value* must be greater than constrained value), etc. As they are not used throughout this dissertation, *Predictions* are not specified further.

By means of this *common vocabulary,* in principle any discipline or domain can be represented semantically. In the context of this dissertation, mediation rules allow for mediating the discipline-specific vocabularies via the *mechatronics* vocabulary with the *common* vocabulary.

5.3.3. Mediation rules: Mediating between the vocabularies

In order to mediate in between the involved vocabularies, respective *mediation rules* need to be employed. Within the following, the mediation between the involved discipline, domain and common vocabularies is discussed. Subsequently, it is discussed how additional mediation can be used, e.g., in order to apply background knowledge to the involved models.

Mediation between discipline, domain and common vocabularies

For the purpose of inconsistency management, it is essential to mediate between the different types of vocabularies used throughout the engineering process. Within the context of this dissertation, the set relations between the vocabularies being used can, according to Figure 5.16, be illustrated as shown in Figure 5.21. As can be seen from this Figure, the vocabularies for the disciplines *mechanics*, *electrics* and *software* are mediated to the *common vocabulary*. Accordingly, a set of mediation rules allows for mediating from the *mechatronics vocabulary* to the *common vocabulary* as well as from the respective *project* and *QUDT vocabularies* to the *common vocabulary*.

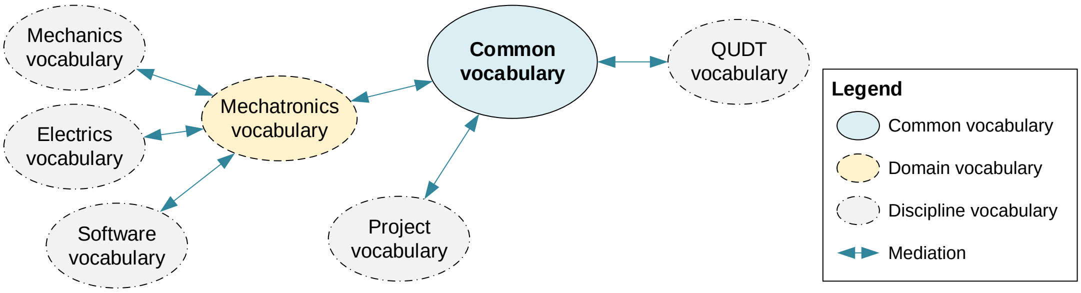


Figure 5.21. Semantic mediation between vocabularies that are focused on in this dissertation

As introduced in Section 5.3.1, the mediation between the respective vocabularies can be defined by means of a set of graph transformation rules. These rules define a mapping between the entities in the specialized vocabularies and the ones in the respective general vocabularies, thereby mapping the concepts from more detailed ones (e.g., discipline-specific concepts) to more general ones (e.g., domain-specific concepts).

Background knowledge mediation

Additionally to applying the concept of mediation for the purpose of mapping between distinct vocabularies, further use cases can be identified. Especially for the purpose of incorporating background knowledge for the respective dependent models, the mediation concept can be applied.

One example for such a background knowledge mediation is the application of configuration rules. If one of the involved background knowledge models comprises of knowledge that is relevant to the consistency of configurations in another domain, mediation rules can be applied for this purpose. Another example is the mediation between different units being used in disparate engineering models. As the QUDT vocabulary defines a conversion factor and offset that defines how one unit is transformed into another one, respective mediation rules can be formulated to mediate all units into a common unit system. Consequently, as can be seen from Equation (5.16), for each *QuantityValue* q, the according *Unit* u is identified. By means of the underlying *QuantityKind* k, the respective basic SI unit nU is retrieved and, based on the *conversionOffset* o and *conversionMultiplier* m, the new value $v \cdot m + o$ is inferred for the newly created *mediatedValue* with the according *mediatedUnit*.

$$\begin{aligned} QuantityValue_Q(q) \wedge unit_Q(q,u) \wedge\ value(q,v) \\ \wedge\ conversionOffset_Q(u,o) \\ \wedge\ conversionMultiplier_Q(u,m) \\ \wedge\ quantityKind_Q(u,k) \wedge\ quantityKind_Q(nU,k) \\ \wedge\ isSIUnit_Q(nU, \text{“}true\text{”}) \rightarrow & \ QuantityValue_Q(nQ) \\ & \wedge\ mediatedUnit_Q(nQ,nU) \\ & \wedge\ mediatedValue_Q(nQ, v*m+o) \qquad (5.16) \end{aligned}$$

An exemplary application of this mediation rule within an RDF graph is illustrated in Figure 5.22. Two *QuantityValues value1* and *value2* are defined, either with the unit *Kilogram* or *Pound*, and with according values. By means of the mediation rule defined in Equation (5.16) as well as the RDF graph shown in Figure 5.22, the *mediatedValues* can be inferred for both *QuantityValues value1* and *value2*.

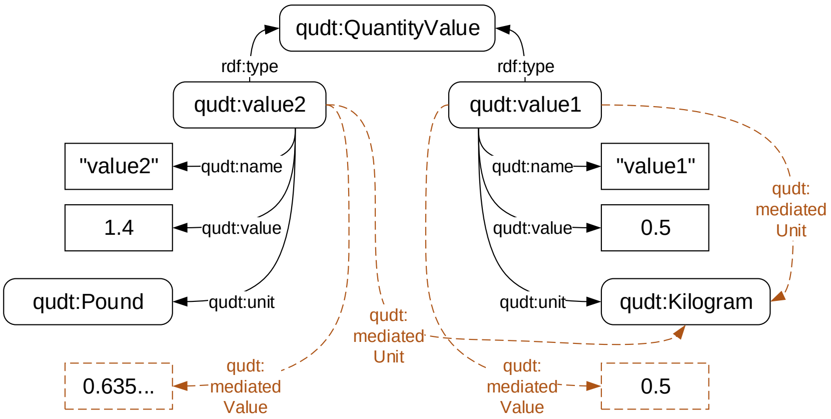


Figure 5.22. Mediation within an exemplary graph using the *qudt* vocabulary (inferred information is denoted in dashed lines)

5.4. Diagnosis and Handling: Diagnosis and Handling of Inconsistencies

Given that RDF is used as the common representational formalism for the models, appropriate means to diagnose and handle inconsistencies can be put in place. Accordingly, the aim of this section is to introduce the means to specify, diagnose and handle inconsistencies by means of the SPARQL Query and Update Languages. First, the conceptual basis for specifying and executing inconsistency diagnosis and handling rules is introduced in the following (Section 5.4.1). As discussed in Chapter 2, four distinct types of inconsistencies are envisioned to be handled by the inconsistency management approach: notational, conventional, correspondence as well as domain-specific inconsistencies. Consequently, it is shown how intra-model inconsistencies can be diagnosed and handled by the inconsistency management framework (Section 5.4.2)- Subsequently, it is shown how inter-model inconsistencies are incorporated in the approach (Section 5.4.3). Finally, the different types of inconsistencies to be considered for the distinct engineering models of automated production systems are introduced in Section 5.4.4

5.4.1. Structuring the Inconsistency Diagnosis and Handling Problem

As discussed in Chapter 3, using a respective knowledge base that captures all the different models used throughout the engineering of automated production systems, a support for specifying and diagnosing different types of inconsistencies (Requirement 2) as well as to handle these inconsistencies (Requirement 3) is necessary. Moreover, to support domain experts – namely, discipline-specific engineers – in identifying the *hot spots* in their system, the means to assess and measure potentially occurring inconsistencies and their handling actions is required (Requirement 4).

In order to structure the inconsistency management problem, an *inconsistency* vocabulary was formulated in a joint research effort together with *Technische Universität Wien* and *Technische Universität München* – the results were published in [Fel+16c]. The extended vocabulary is illustrated in Figure 5.23.

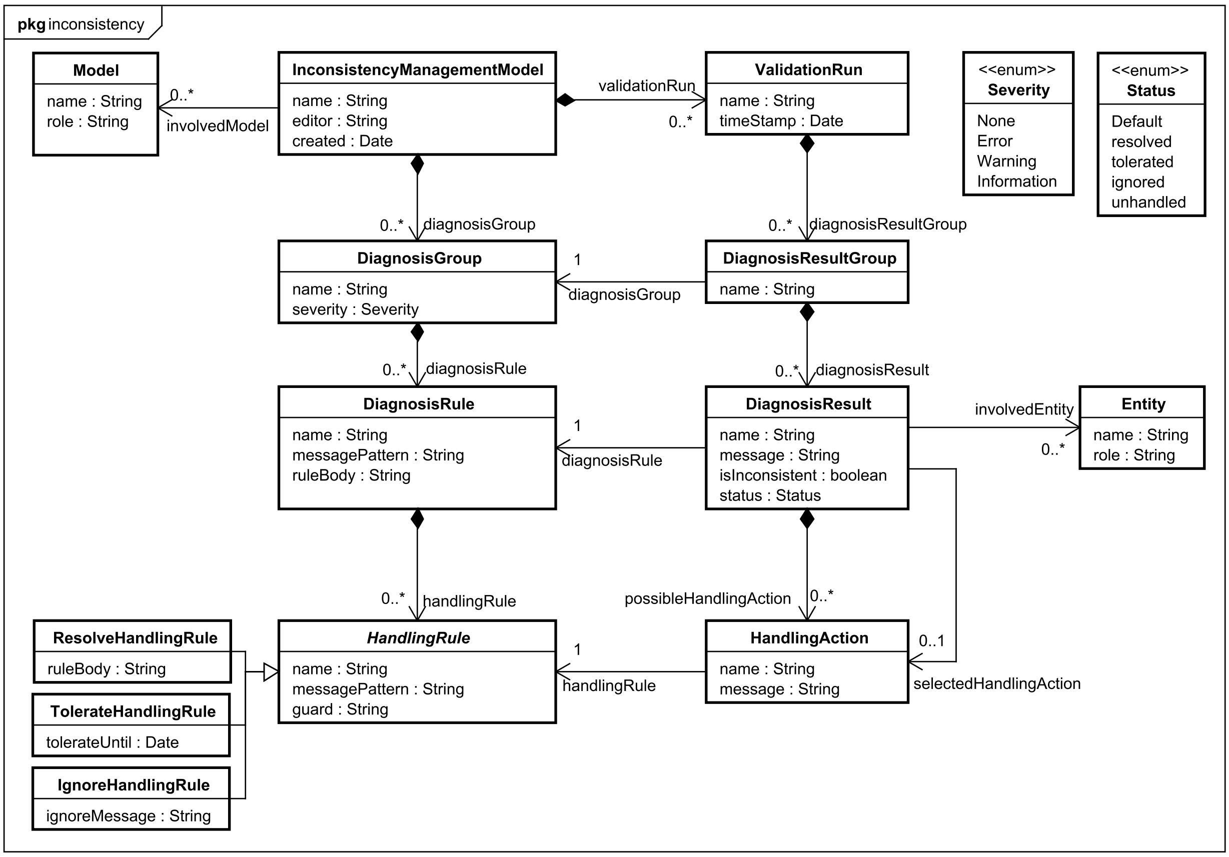


Figure 5.23. Overview of the inconsistency management vocabulary [extended from Fel+16c]

Therein, an *InconsistencyManagementModel* structures the entire inconsistency management problem. Within a *DiagnosisGroup*, a group of inconsistency *DiagnosisRules* is defined with an according *name* and *severity* (e.g., *Error*, *Warning* or *Information*). Accordingly, a *DiagnosisRule* is represented through a *name*, a *messagePattern*, which defines what the message to be displayed to the user is, and a *ruleBody* that includes the actual inconsistency diagnosis rule. Additionally to diagnosing inconsistencies, handling them is supported by *HandlingRules* – which can either be *ResolveHandlingRules*, *TolerateHandlingRules* or *IgnoreHandlingRules.* Accordingly, *HandlingRules* are specified through their *name*, an according *messagePattern* and an optional *guard.* Based on such a *guard*, it is decided whether or not a *HandlingRule* is applicable to the diagnosed inconsistency or not.

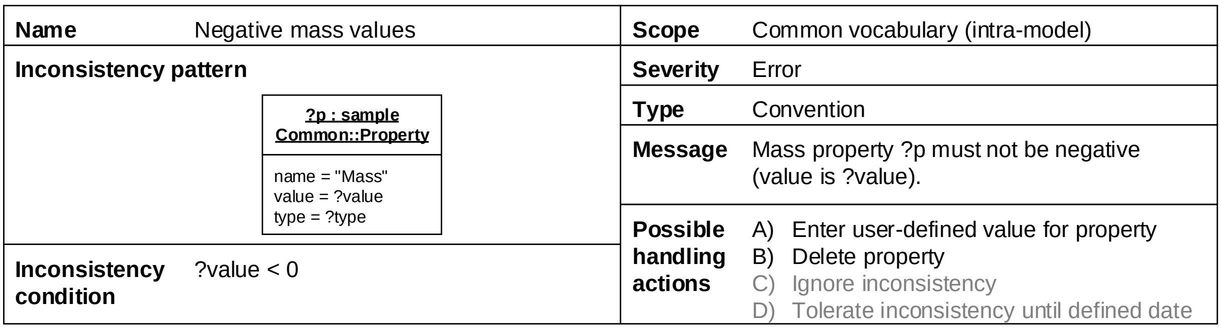
Whereas the previously discussed components describe how the *DiagnosisRules* and *HandlingRules* are specified, the *inconsistency* vocabulary also includes a specification of *ValidationRuns*. These *ValidationRuns* represent executions of the specified rules against the engineering models-Hence, a *DiagnosisGroup* results into a *DiagnosisResultGoup* in a *ValidationRun*; a *DiagnosisRule* results into a *DiagnosisResult*, which defines whether a pattern match resulted into a match that *isInconsistent* or not and which defines the current *status* of a diagnosed inconsistency; a *HandlingRule* is represented through a *HandlingAction*. Consequently, the user selects a *HandlingAction* from the *possibleHandlingActions*, which is to be executed during the next *ValidationRun*.

By means of that *inconsistency* vocabulary, the entire inconsistency management problem is also represented as an RDF graph – hence, it is also part of the model knowledge base. Consequently, any software program (e.g., a discipline-specific engineering tool) that accesses the knowledge base can also diagnose and handle the respective inconsistencies in the models. Hence, by means of the *inconsistency* vocabulary, an interactive approach is enabled that allows users to decide on how to react on diagnosed inconsistencies as well as to document the decisions being made.

In order to specify *inconsistency diagnosis rules* as well as *inconsistency handling rules*, we make use of the SPARQL Query and Update Languages. Consequently, inconsistency diagnosis rules can be expressed by means of patterns that are matched against the knowledge base (Query Language). Accordingly, inconsistency handling rules are either defined (1) as a resolve handling rule, which replaces the inconsistent pattern (Update Language), (2) as a tolerate handling rule that tolerates the diagnosed inconsistency until a specific date and (3) as an ignore handling rule, which ignores the diagnosed inconsistency.

5.4.2. Specifying Intra-model Inconsistencies

Intra-model inconsistencies refer to inconsistencies that occur within a single model. Hence, links between distinct models are not incorporated in these types of inconsistencies. A typical example for such an intra-model inconsistency is illustrated in Figure 5.24 and refers to the demand that mass values must not be negative. Therein, the *inconsistency pattern* is illustrated graphically to denote, which graph elements are potential candidate for the inconsistency. Furthermore, the *inconsistency condition* describes, what condition must hold to consider the graph elements as inconsistent. Further on, respective meta information is visualized such as the intended scope of the inconsistency, its expected severity as well as a message that should be generated in case an inconsistency is diagnosed. Accordingly, different handling actions are listed. This example is used throughout the following to illustrate, how SPARQL is used to specify, diagnose and handle intra-model inconsistencies.



Name	Negative mass values	**Scope**	Common vocabulary (intra-model)

Inconsistency pattern	?p : sample Common::Property
	name = "Mass"
	value = ?value
	type = ?type
		Severity	Error

		Type	Convention
		Message	Mass property ?p must not be negative (value is ?value).
Inconsistency condition	?value < 0	**Possible handling actions**	A) Enter user-defined value for property B) Delete property C) Ignore inconsistency D) Tolerate inconsistency until defined date

Figure 5.24. Intra-model inconsistency in the *sampleCommon* vocabulary

Diagnosing Intra-model Inconsistencies

As discussed beforehand, the SPARQL Query Language is used to specify intra-model inconsistency diagnosis rules. Consequently, the affected entities can be retrieved by an inconsistency pattern, which specifies, what conditions must be met in order for an inconsistency to be present. The respective SPARQL query makes use of the vocabularies, in which the inconsistency is assumed to occur. The inconsistency diagnosis rule that can be derived from the example in Figure 5.24 is introduced in Figures 5.17 and 5.18. This query makes use of the mediated information to identify, which mass properties have negative values. From this diagnosis rule, the basic structure of an inconsistency diagnosis rule formulated by means of the SPARQL Query Language is apparent: For one, the variables defined in the SPARQL SELECT Query include (1) the entities that are involved within the inconsistency, and (2) an *isInconsistent* variable that returns whether an inconsistency occurs (TRUE) or not (FALSE). Moreover, such a SPARQL query includes an *inconsistency pattern* part, which defines the pattern to be matched against the RDF graph. In the example illustrated in Listing 5.1, the pattern matches any $Property_C$ with its according $value_C$ and $type_C$ that is named *"Mass"*. If the pattern finds a match, the result of the mathematical comparison $?value < 0$ is bound to the variable *isInconsistent* – by that, it can be concluded whether the mass property is inconsistent or not.

```
PREFIX sampleCommon: <http://www.example.org/sampleCommon#>

SELECT ?entity ?property ?value ?type ?isInconsistent WHERE {
  #Inconsistency pattern
  ?entity sampleCommon:property [
    a sampleCommon:Property ;
      sampleCommon:name "Mass" ;
      sampleCommon:value ?value ;
      sampleCommon:type ?type
    ] .

  #Inconsistency condition
  BIND ( ( ?value < 0 ) AS ?isInconsistent ) .
}
```

Listing 5.1 Exemplary inconsistency diagnosis rule to ensure that all mass properties have positive values formulated as SPARQL SELECT query

Handling Intra-model Inconsistencies

In order to resolve the diagnosed inconsistency, as denoted in Figure 5.24, the user can (besides tolerating and ignoring the inconsistency) either define a new value for an inconsistent property or delete the inconsistent value. Hence, two distinct resolution rules are defined – one for introducing a new, user-defined value (see Listing 5.2) and one for deleting the value (see Listing 5.3). As these handling actions are generated for specific inconsistency diagnosis results – i.e., for each inconsistency that has been diagnosed from the respective diagnosis rule – they actions must be regarded as templates. Hence, for each identified inconsistent *entity*, a possible handling action is generated and the place-holders for, e.g., the entity's Uniform Resource Identifier (URI) (see *\$?entity\$*) and the user-defined value (see *\$?newValue\$*) are replaced. Throughout the *originalValueTriple*, it can be identified, which value must be changed in order to replace the original (non-mediated) value. Although not illustrated here, respective additional information can be added to the handling rules. For instance, names and severity values can be given according to Figure 5.23 in order to support users in identifying the respective appropriate handling actions. Further on, message patterns are defined accordingly, which make use of the attributes in the SPARQL queries, and are, therefore, automatically generated during the execution process.

```
PREFIX sampleCommon: <http://www.example.org/sampleCommon#>

DELETE {
  ?subject ?predicate ?object .
} INSERT {
  ?subject ?predicate $?newValue$ .
} WHERE {
  $?entity$ sampleCommon:property ?property .
  ?property sampleCommon:originalValueTriple [
    sampleCommon:subject ?subject ;
    sampleCommon:predicate ?predicate ;
    sampleCommon:object ?object
  ] .
}
```

Listing 5.2 Exemplary inconsistency handling rule that allows users to enter user-defined values formulated as SPARQL Update action

```
PREFIX sampleCommon: <http://www.example.org/sampleCommon#>

DELETE {
  ?subject ?predicate ?object .
} INSERT {

} WHERE {
  $?entity$ sampleCommon:property ?property .
  ?property sampleCommon:originalValueTriple [
    sampleCommon:subject ?subject ;
    sampleCommon:predicate ?predicate ;
    sampleCommon:object ?object
  ] .
}
```

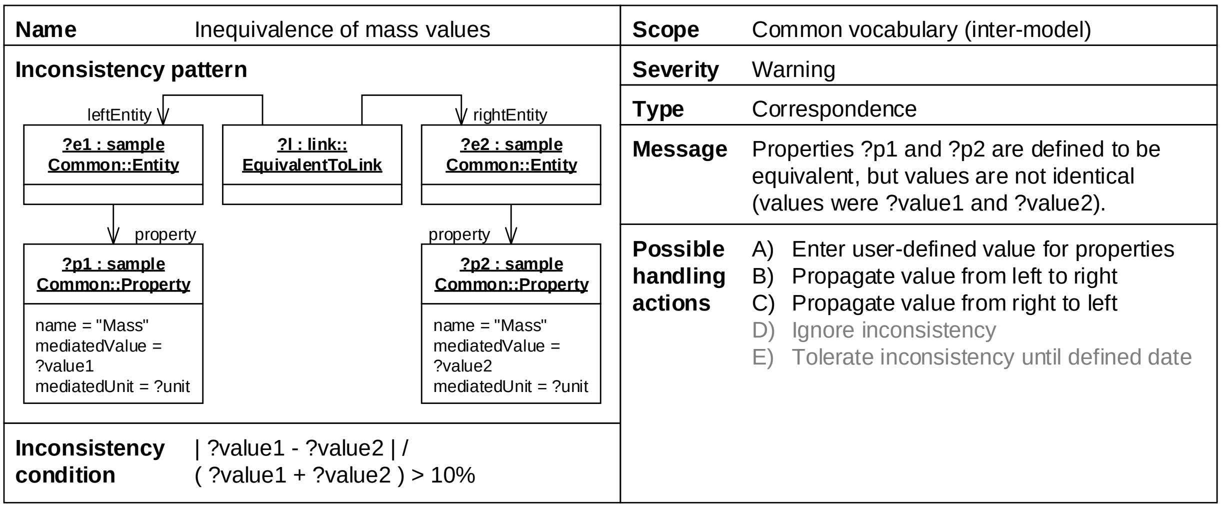
Listing 5.3 Exemplary inconsistency handling rule that deletes inconsistent values formulated as SPARQL Update action

5.4.3. Specifying Inter-model Inconsistencies

Whereas intra-model inconsistencies incorporate a single model to draw conclusions on whether an inconsistency occurs or not, inter-model inconsistencies refer to correspondence rules that incorporate two or more models. Hence, links between distinct models must be incorporated for these types of inconsistencies. A typical example for such an inter-model inconsistency is illustrated in Figure 5.25. Therein, *EquivalentToLinks* between different *Entities* are taken into consideration. If both entities that are said to be equivalent to each other have a mass property, both of them must be equivalent (or in this case: they must not deviate from each other by more than 10%). This example is used throughout the following to illustrate, how SPARQL is used to specify, diagnose and handle inter-model inconsistencies.

Diagnosing Inter-model Inconsistencies

Analogously to intra-model inconsistencies, SPARQL queries are used for the purpose of specifying inter-model inconsistency diagnosis rules. Accordingly, an inconsistency pattern is used to retrieve



Name	Inequivalence of mass values	**Scope**	Common vocabulary (inter-model)
Inconsistency pattern		**Severity**	Warning
		Type	Correspondence
		Message	Properties ?p1 and ?p2 are defined to be equivalent, but values are not identical (values were ?value1 and ?value2).
		Possible handling actions	A) Enter user-defined value for properties B) Propagate value from left to right C) Propagate value from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date
Inconsistency condition	| ?value1 - ?value2 | / (?value1 + ?value2) > 10%		

Figure 5.25. Inter-model inconsistency in the *sampleCommon* vocabulary

the potentially inconsistent parts of the graph. In contrast to the previous intra-model inconsistency, this inconsistency diagnosis rule must incorporate the mediated values and units of the respective properties, as linked entities may initially have different units. The resulting mediated values *value1* and *value2* are retrieved and, by means of the mathematical formula $|value1 - value2|/(value1 + value2)$ it is determined, whether the values have a deviation that is greater than 10% or not. Accordingly, any result of executing this SPARQL query for which *isInconsistent* is determined to be true is considered to be inconsistent.

```
PREFIX sampleCommon: <http://www.example.org/sampleCommon#>
PREFIX qudt: <http://www.example.org/qudt#>
PREFIX link: <http://www.example.org/link#>

SELECT ?link ?left ?right ?value1 ?value2 ?isInconsistent WHERE {
  #Inconsistency pattern
  ?link a link:EquivalentToLink ;
    link:leftEntity [ link:entityReference ?left ] ;
    link:rightEntity [ link:entityReference ?right ] .

  ?left sampleCommon:property [ sampleCommon:name "Mass" ;
      qudt:mediatedValue ?value1 ; qudt:mediatedUnit ?unit ] .
  ?right sampleCommon:property [ sampleCommon:name "Mass" ;
      qudt:mediatedValue ?value2 ; qudt:mediatedUnit ?unit ] .

  #Inconsistency condition
  BIND ( ( ABS(?value1 - ?value2) / ( ?value1 + ?value2 ) > 0.1 ) AS ?
      isInconsistent ) .
}
```

Listing 5.4 Exemplary inconsistency diagnosis rule to ensure that all entities said to be equivalent to each other have similar mass values formulated as SPARQL SELECT query

Handling Inter-model Inconsistencies

As illustrated in Figure 5.24, besides tolerating and ignoring the diagnosed inconsistencies, there are three strategies to resolve the inconsistency: (1) to replace the values of both linked entities with a user-defined one, (2) to propagate the left entity's value to the right one, and (3) to propagate the right entity's value to the left one. However, all of these three strategies can be covered by a single handling action that must be configured accordingly (see Listing 5.5). Therein, the *WHERE* part of the SPARQL Update action not only identifies the *originalValueTriples* of the respective properties, but also the original units being used for these values. By means of these units, it can be calculated, which value needs to be bound to which entity – hence, the user chooses the correct handling action based on the mediated values and the initial values are updated accordingly. If values should be propagated from one entity to another, the *$userValue$* place-holder is replaced accordingly.

```
PREFIX sampleCommon: <http://www.example.org/sampleCommon#>
PREFIX qudt: <http://www.example.org/qudt#>
PREFIX link: <http://www.example.org/link#>

DELETE {
  ?subjectLeft ?predicateLeft ?objectLeft .
  ?subjectRight ?predicateRight ?objectRight .
} INSERT {
  ?subjectLeft ?predicateLeft ?newLeft .
  ?subjectRight ?predicateRight ?newRight .
} WHERE {
  $?left$ sampleCommon:property ?propertyLeft .
  ?propertyLeft sampleCommon:originalValueTriple [
    sampleCommon:subject ?subjectLeft ;
    sampleCommon:predicate ?predicateLeft ;
        sampleCommon:object ?objectLeft ] ;
    sampleCommon:type ?leftUnit .
    ?leftQudtUnit qudt:symbol ?leftUnit ; qudt:conversionOffset ?leftOffset ; qudt:conversionMultiplier ?leftMultiplier .
  $?right$ sampleCommon:property ?propertyRight .
  ?property sampleCommon:originalValueTriple [
    sampleCommon:subject ?subjectRight ;
    sampleCommon:predicate ?predicateRight ;
    sampleCommon:object ?objectRight ] ;
    sampleCommon:type ?rightUnit .
    ?rightQudtUnit qudt:symbol ?rightUnit ; qudt:conversionOffset ?rightOffset ; qudt:conversionMultiplier ?rightMultiplier .
  BIND ( $?userValue$ AS ?newValue ) .
  BIND ( ( ( ?newValue - ?leftOffset ) / ?leftMultiplier ) AS ?newLeft ) .
  BIND ( ( ( ?newValue - ?rightOffset ) / ?rightMultiplier ) AS ?newRight ) .
}
```

Listing 5.5 Exemplary inconsistency handling rule that allows users to enter user-defined values formulated as SPARQL Update action

5.4.4. Specifying, Diagnosing and Handling of Different Types of Inconsistencies

As discussed in Chapter 2, four types of inconsistency rules are to be distinguished, which are introduced in the following for the different engineering models in the automated production systems domain.

Notational inconsistency rules

Notational inconsistency rules refer to inconsistencies that result from the definition of the modelling language that is used for the respective model. Although not specifically emphasized in this dissertation, especially as most modelling tools already support the diagnosis and handling of notational inconsistency rules, these notational inconsistency rules can be specified by means of the presented formalism. Examples for such notational inconsistency rules are, e.g., definitions on what entity is to be used for a certain attribute or value restrictions for certain properties. In most cases, the resolution of those notational inconsistency rules involves, e.g., adding user-defined attributes or deleting certain relations in the models.

Conventional inconsistency rules

Conventional inconsistency rules involve common conventions that cannot be captured explicitly within the modelling language's abstract syntax. Such common conventions comprise, e.g., naming conventions or conventions on which entities or types must be used under specific circumstances. Consequently, these conventions can often be derived from, e.g., company- or project-specific design guidelines and heuristics. No common resolution rule can be given for such conventional rules – in some cases, changing an attribute's value (e.g., a name) to a user-specific one can resolve the inconsistency.

Correspondence inconsistency rules

Correspondence inconsistency rules refer to rules that specify, how different entities in disparate models must relate to each other. Hence, these correspondence inconsistency rules refer to how disparate models are related to each other by means of the *link* vocabulary.

In case such a correspondence inconsistency is diagnosed, three distinct cases can be distinguished:

- *Link is valid*: In case the link between the entities is valid, no handling action must be taken into account.

- *Link is not valid*: In case the two entities are linked, but the link is not valid (e.g., because the entities are not named equivalently), respective handling actions to improve the link are generated.

- *Link does not exist*: In case the two entities are not linked, a respective link must be established. In this case, either the user must select the two entities to be linked, or a handling action is taken to generate a link based on a pre-defined heuristic (e.g., equivalently named entities).

Domain-specific inconsistency rules

Domain-specific inconsistency rules refer to rules that are applicable for a specific domain, e.g. for the automated production systems domain. These *domain-specific inconsistency rules* often involve background knowledge models, which are applied to the engineering models, e.g., regarding unit conversions or certain configurations.

5.5. Summary

In this chapters, the framework for managing inconsistencies – that is, for diagnosing and handling inconsistencies – in engineering models of the automated production systems domain is presented.

By means of a knowledge base that incorporates both the engineering models and background knowledge models, formulated by means of RDF, a common representational formalism is given (Requirement 1.1). The rule-based mediation mechanism allows for transforming between this knowledge base and common semantic concepts (Requirement 1.2), that result, e.g., from links in between these models (Requirement 1.3). Finally, it is shown how the SPARQL Query and Update Languages are used for the purpose of specifying inconsistency diagnosis and handling actions (Requirements 2 and 3), which can be executed against the central knowledge base.

As these diagnosis and handling rules are further annotated with information such as the severity as well as involved entities of an inconsistency, a basis to assess the *hot spots* within the engineering models is given (Requirement 4). While developing the framework, solely standard technologies from the Semantic Web initiative, namely RDF, RDFS and SPARQL, are applied. Consequently, the means to extend the entire framework towards further domains and applications is given (Requirement 5).

Chapter 6.

Evaluation: Assessment and Comparison of the Applicability

To validate, whether the presented framework is capable of managing inconsistencies in engineering models of automated production systems, this chapter aims at assessing the basic feasibility and performance of the approach based on the requirements specified in Chapter 3.

The evaluation is structured into several parts to validate the feasibility of the proposed incon-

Table 6.1. Overview of the evaluation strategy in this dissertation

Req.	**Name**	**Evaluation Strategy**	**Sect.**
1	Model knowledge base	The knowledge base is evaluated by applying the approach to the disparate models of two distinct application examples – a lab-scale and an industry-style one – and by validating, whether the models can be represented by means of the model knowledge base.	6.3, 6.4
2	Inconsistency diagnosis	For the application examples, the different types of inconsistencies are introduced by means of representative examples. It is validated, whether all types of inconsistencies can be diagnosed by means of the inconsistency management approach.	6.3, 6.4
3	Inconsistency handling	For the application examples, the different types of handling actions that serve the purpose of handling the diagnosed inconsistencies are introduced by means of representative examples. It is validated, whether all types of handling actions can be executed by means of the inconsistency management approach.	6.3, 6.4
4	Measurement and assessment capabilities	For validating, whether the approach is able to measure the diagnosed inconsistencies and assess the impact of the different handling actions, the approach is applied to the two application examples.	6.3, 6.4
5	Operationalization	For the purpose of gaining insights into the operationalization of the inconsistency management approach, it is compared to another inconsistency management approach. Furthermore, by means of insights into the performance and scalability of the approach, its applicability to larger models is estimated.	6.5, 6.6

sistency management framework (see Table 6.1). As a basis to perform these different steps, a prototypical software implementation of the proposed concept is introduced in Section 6.1. At the hand of a central case study (Section 6.2) and two according application examples – a lab-scale one (Section 6.3) that illustrates how the types of inconsistencies described in Chapter 5 are managed and an industry-style one (Section 6.4) that illustrates how typical background knowledge in the automated production systems domain can be considered for inconsistency management – the feasibility of the concept is evaluated. Moreover, a comparison of the proposed framework with the one presented in [Fel+16c] is done at the hand of representative engineering models (see [Fel+15a]) in Section 6.5. With the intent to assess the capability to operationalize the approach within practical applications, the performance and scalability that can be achieved with the proposed framework is discussed in Section 6.6. The results obtained from the evaluation are summarized in Section 6.7. An overview of the evaluation strategy, especially with regard to the requirements specified in Chapter 3, is given in Table 6.1.

6.1. Prototypical Software Implementation

To perform the different evaluation stages, a prototypical software implementation is provided. It should be noted that the conception, implementation and evaluation of this software prototype were conducted as part of a Master's Thesis at *Technische Universität München* – a detailed documentation of the software prototype is published in [Hau16; Fel+17].

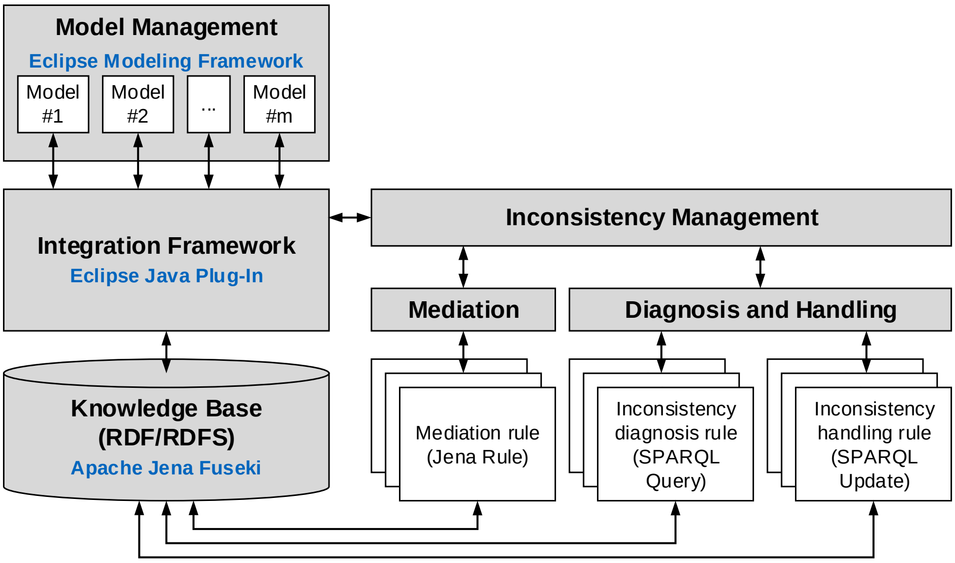

Model Management
Eclipse Modeling Framework
Model #1
Model #2
...
Model #m
Integration Framework
Eclipse Java Plug-In
Inconsistency Management
Mediation
Diagnosis and Handling
Knowledge Base (RDF/RDFS)
Apache Jena Fuseki
Mediation rule (Jena Rule)
Inconsistency diagnosis rule (SPARQL Query)
Inconsistency handling rule (SPARQL Update)

Figure 6.1. Assignment of the architectural components of the concept to respective software components

Figure 6.1 shows the different software components that are used to realize the different parts of the entire inconsistency management framework. As a basis for the entire framework, the Eclipse Modeling Framework (EMF) [Ecl16c] is used. Specifically, for each of the involved vocabularies of the presented concept, a respective metamodel was formulated by means of Ecore – a pragmatic implementation of the Meta Object Facility (MOF) [OMG15a] that allows for creating and maintaining metamodels as well as respective model instances. Respectively, for each of the exemplary models, Extensible Markup Language (XML) Metadata Interchange (XMI) model instances can be specified by means of this framework. The mapping between both the Ecore metamodels and XMI model instances as well as Resource Description Framework (RDF)/RDF Schema (RDFS) is done through MOF Model To Text (MOFM2T) transformations specified in the respective standard [Obj08] and executed by means of the Eclipse Acceleo plug-in [Ecl16a]. The central integration framework is realized as an Eclipse Java plug-in, which coordinates the respective dependent software components. To allow for capturing the knowledge base, the Apache Jena Fuseki triple store [Apa16] is applied, which also serves as the endpoint to execute the inconsistency diagnosis and handling rules by means of SPARQL Protocol and RDF Query Language (SPARQL).

6.2. Introduction to the Case Study: The Pick-and-Place Unit

To evaluate the feasibility of the presented inconsistency management framework, a case study is used throughout the following. In particular, the so-called Pick and Place Unit (PPU) [Vog+14b] is used – a lab-scale case study that is developed and maintained at *Technische Universität München*. Therein, the PPU is not only a physical automated production systems, but rather stands for a set of scenarios that describe the *evolution* of an automated production system, specifying the changes that are applied throughout the parallel or sequential evolution of an automated production system. In particular, within the evaluation part of this dissertation, the change from scenario 12 to scenario 13 is investigated, which – from an inconsistency management perspective – describes an appropriate basis for evaluation [Fel+15a].

The PPU consists of typical parts of an automated production system and is controlled by a single Programmable Logic Controller (PLC). An overview of the PPU is given in Figure 6.2. Therein, white and black plastic as well as metallic work pieces (WPs) are processed. In the *Stack* module, WPs are stored and then pushed separately to the handover position of a *Crane*. The *Crane* picks the WP with a vacuum gripper, lifts them, and rotates them – depending on the WP type – either to an outlet *Ramp*, which is located at an angle of 90° to the handover position or to a *Stamp* module, which is located at an angle of 180° to the handover position. In the *Stamp* module, metallic WPs are labelled by putting a specific pressure on the WP using a stamping cylinder. Within the outlet *Ramp*, WPs are stored. Although the PPU represents only a fraction of an industrial automated production system, it contains a multitude of mechanical, electrical and software components and, thus, has an appropriate level of complexity for the purpose of describing different application examples in the context of evaluating an inconsistency management framework.

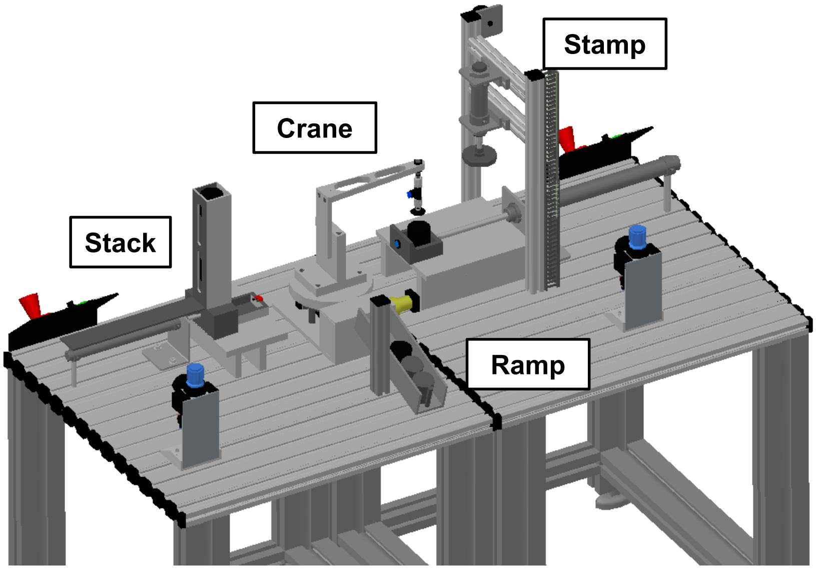
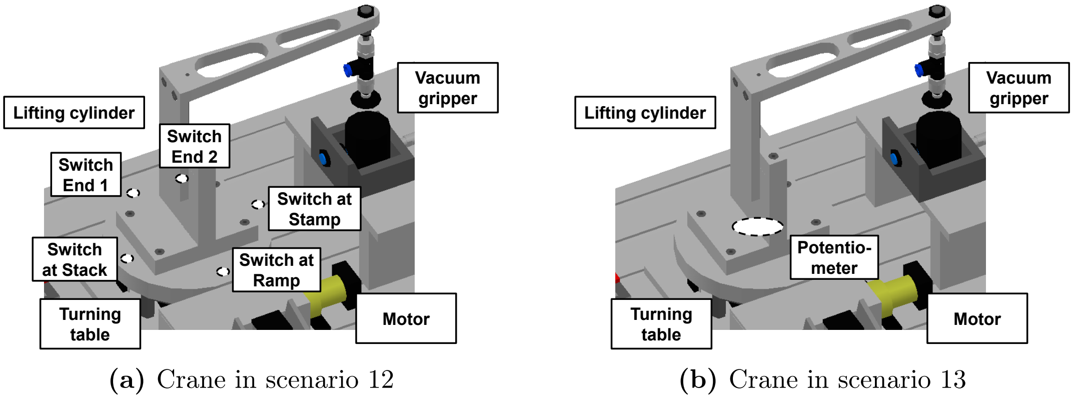


Figure 6.2. Overview of the PPU case study (excerpt from [Vog+14b])

Within the case study, a special emphasis is put on the *Crane* module. As depicted in Figure 6.3, the *Crane* module consists of a *Lifting Cylinder* to raise and lower WPs, a *Vacuum Gripper* that allows to grip and release WPs as well as a *Turning Table* for turning WPs from one module to another. In an initial scenario 12, five *Micro Switches* are attached to the bottom of the *Turning Table* – three of them for detecting the current positions of the *Crane* at the *Modules Stack*, *Ramp* and *Stamp* as well as two of them to detect the *End Positions* of the *Module*. Although these *Micro Switches* are sufficient for an initial scenario, they do not provide the means to, e.g., stop the *Crane* module at a position in between the distinct *Micro Switches*. Consequently, in scenario 13, a *Potentiometer* allows for measuring the current angular position of the *Crane* – and, hence, provides the means to stop the module at any angular position. This evolutionary transition from allowing a sole discrete positioning of WPs to enable a continuous positioning is investigated as one example for inconsistency management throughout this section.



(a) Crane in scenario 12

(b) Crane in scenario 13

Figure 6.3. Overview of the crane in the PPU case study (excerpt from [Vog+14b])

6.3. Evaluation Stage 1: Evaluation of the Feasibility for a Lab-Scale Application Example

To validate the feasibility of the proposed framework for managing inconsistencies in heterogeneous models of automated production systems, this evaluation stage makes use of the information provided in [Vog+14b] and maps the respective information to the metamodels and models introduced in Chapter 5. Based on the resulting models, the introduced inconsistencies are diagnosed and handled. In the following, the incorporated models are described in Section 6.3.1. Subsequently, the different inconsistencies to be diagnosed and handled are described in Section 6.3.2. The results of this evaluation stage are summarized in Section 6.3.3.

6.3.1. Overview of the Incorporated Models

As a basis for the lab-scale application example, the respective models using the vocabularies introduced in Chapter 5 are created:

- A *project model* defines the overall project set-up of the PPU case study. Therein, besides the overall project data, the for the overall example is defined (see Figure 6.4a).

- The *mechanics model* defines the physical structure of the PPU. Besides the mechanical parts being used for the project, interfaces between the respective parts are specified (see Figure 6.4b).

- Within the *electrics model*, the electrical wirings between the controller, respective bus terminals and the electrical parts are specified (see Figure 6.4c).
- A *software model* specifies the software dependency model for the PPU (see Figure 6.4d).

Please note that the models denoted in Figure 6.4 serve illustrative purposes and do not follow a standard notation. All of these models are specified based on the technical documentation that is available for the PPU case study in [Vog$^+$14b]. Within this section, it is focused on the *Crane* module of the PPU.

6.3.2. Diagnosing and Handling the Inconsistencies

For the lab-scale evaluation example, a set of intra-model and inter-model inconsistencies is investigated through this evaluation stage. Certainly, these types of inconsistencies do not cover all possible inconsistencies that can exist in the distinct heterogeneous models – however, these inconsistencies are regarded to be sufficient for the purpose of evaluating the feasibility of the inconsistency management framework. In particular, inconsistencies in the different categories that are relevant to inconsistency management are defined – namely notational, conventional, correspondence and domain-specific inconsistencies – and introduced in the following. For simplicity reasons, only a rough description of the different inconsistencies is illustrated in the following – a detailed overview of all inconsistencies can be found in Appendix A.

Intra-model inconsistencies

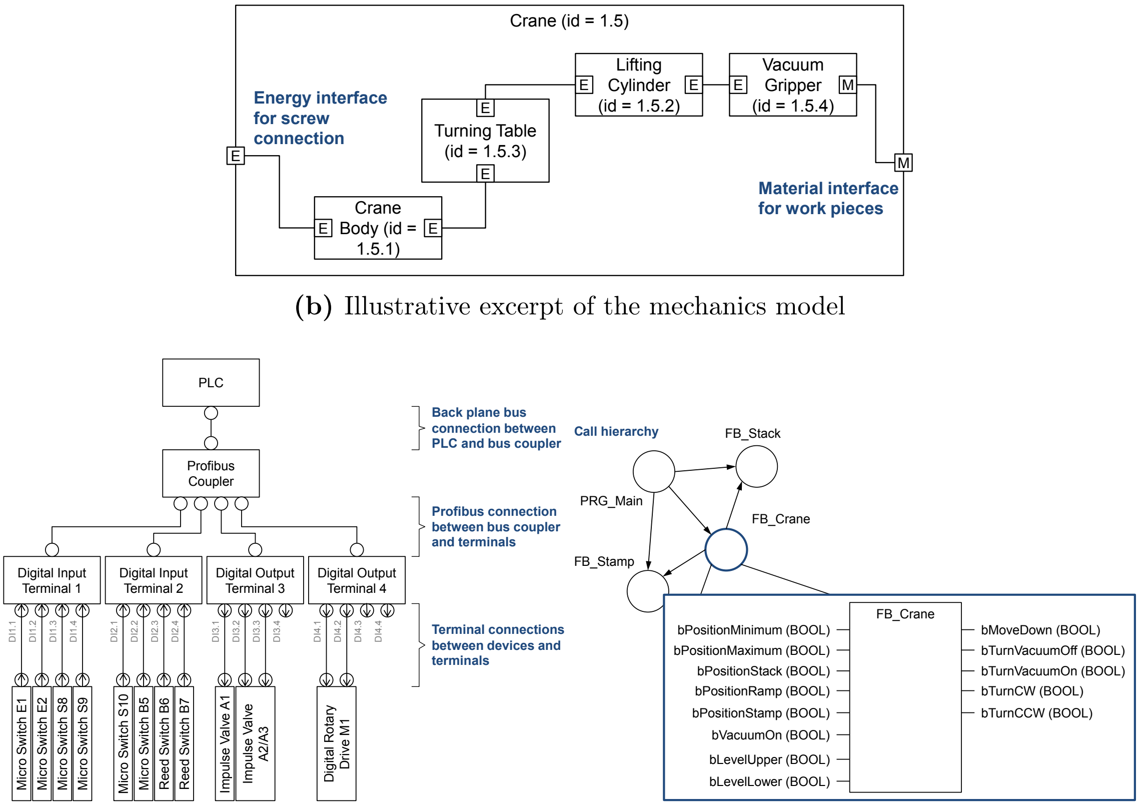
To show the feasibility of the inconsistency management framework for identifying and handling intra-model inconsistencies, 11 distinct types of inconsistencies are defined (see Table 6.2). Therein, besides conventional inconsistencies that describe naming conventions on the *common* vocabulary level (see inconsistencies 1.1 and 1.2 in Table 6.2) as well as on the *mechatronics* vocabulary level (see inconsistencies 1.3 and 1.4 in Table 6.2), further intra-model inconsistencies that are specific for single domains are formulated. For the *project* vocabulary, it is demanded that – from a notation perspective – start and end dates are consistently defined (see inconsistency 1.5). Analogously, for the *mechanics* vocabulary, is is ensured that all parts' IDs are consistently defined (see inconsistency 1.6) – that is: they must be unique, be consecutive and reflect the part hierarchy. For the *electrics* vocabulary, inconsistencies are specified to ensure that interfaces are wired correctly (see inconsistency 1.7) and according to the maximum number of participants specified for the respective couplers and terminals (see inconsistency 1.8). Analogously, within the *software* vocabulary, is is demanded that naming conventions are met (see inconsistency 1.9) and a tree call hierarchy is used (see inconsistency 1.10). Finally, within the evaluation example, global variables are used for mapping sensors and actuators to respective hardware addresses – hence, to ensure that all sensors and actuators are considered within the software programme, it is ensured that all sensors are read (respectively: actuators are written) at least once within the software programme (see inconsistency 1.11).

Inter-model inconsistencies

Accordingly, to evaluate the feasibility of the presented inconsistency management framework regarding its capability to identify and handle inter-model inconsistencies, 4 distinct types of inconsistencies are defined in Table 6.3. Therein, it is defined that each documentation specified in the *project* vocabulary must be linked to an according model in the *mechanics*, *electrics*, and *software* vocabularies. By means of this inconsistency, it is ensured that a holistic specification is available for the engineering project. Moreover, to ensure that all *parts* in the *mechanics* vocabulary as well as all *automation hardware* in the *electrics* vocabulary are appropriately documented in the Bill of

Project overview		**Bill of Material**			
Project name	PPU project	**Name**	**Description**	**ID**	**Manufacturer**
Customer	Customer 1	Crane	Crane module	1.5	Manufacturer 1
Start	01/01/2017	Crane Body	Crane's body	1.5.1	Manufacturer 1
End	12/31/2017	Lifting Cylinder	Crane's lifting cylinder	1.5.2	Manufacturer 1
Environment humidity [%]	80	Turning Table	Crane's rotary platform	1.5.3	Manufacturer 1
		Vacuum Gripper	Crane's vacuum gripper	1.5.4	Manufacturer 1
Environment temperature [°C]	35	E1	Crane at minimum position (micro switch)	1.5.E1	Manufacturer 2
		E2	Crane at maximum position (micro switch)	1.5.E2	Manufacturer 2
		S8	Crane at stack / 0 degrees (micro switch)	1.5.S8	Manufacturer 2
		S9	Crane at ramp / 90 degrees (micro switch)	1.5.S9	Manufacturer 2
		S10	Crane at ramp / 180 degrees (micro switch)	1.5.S10	Manufacturer 2
		B5	Vacuum on (micro switch)	1.5.B5	Manufacturer 2
		B6	Crane in upper position (reed switch)	1.5.B6	Manufacturer 2
		B7	Crane in lower position (reed switch)	1.5.B7	Manufacturer 2
		A1	Move crane down (impulse valve)	1.5.A1	Manufacturer 2
		A2/3	Turn vacuum on / off (impulse valve)	1.5.A2/3	Manufacturer 2
		M1	Crane motor (rotary drive)	1.5.M1	Manufacturer 2
		DI1	Digital input terminal with 4 slots	1.5.DI1	Manufacturer 3
		DI2	Digital input terminal with 4 slots	1.5.DI2	Manufacturer 3
		DO1	Digital output terminal with 4 slots	1.5.DO1	Manufacturer 3
		DO2	Digital output terminal with 4 slots	1.5.DO2	Manufacturer 3

(a) Illustrative excerpt of the project model



(b) Illustrative excerpt of the mechanics model

(c) Illustrative excerpt of the electrics model

(d) Illustrative excerpt of the software model

Figure 6.4. Illustrative overview of the models incorporated in the lab-scale evaluation of the PPU case study

Material (BOM), inconsistencies 1.13 and 1.14 demand that such links are available in the models. Finally, to ensure that electrical sensors and actuators are appropriately reflected in the *software* vocabulary, inconsistency 1.15 in Table 6.3 demands that each wired terminal interface must be represented by a corresponding global software variable.

Table 6.2. Intra-model inconsistencies investigated in evaluation stage 1

No.	**Type**	**Description**	**Vocabulary**	**Reference**
1.1	Convention	Each entity in any vocabulary must follow a pre-defined naming convention – only letters, numbers, or an underscore ('_') should be used.	Common	Figure A.1
1.2	Convention	Each property in any vocabulary must follow a pre-defined naming convention – only letters, numbers, or an underscore ('_') should be used.	Common	Figure A.2
1.3	Convention	Each Element in the *mechatronics* vocabulary must be named in *UpperCamelCase* convention.	Mechatronics	Figure A.3
1.4	Convention	Each Attribute in the *mechatronics* vocabulary must be named in *LowerCamelCase* convention.	Mechatronics	Figure A.4
1.5	Convention	A project's start and end dates must be consistent to each other – that is: the end date must lie after the start date.	Project	Figure A.5
1.6	Convention	A part's ID must (i) be unique, (ii) be consecutive, and (iii) reflect the part hierarchy – that is: if *partA* has ID *A*, its child part *partB* must have an ID similar to *A_B*.	Mechanics	Figure A.6
1.7	Convention	Interfaces must be wired correctly – that is: (i) in case of "simple" interfaces they must be wired from in- to out-interfaces or vice versa and (ii) in case of "complex" bus interfaces they must be wired to interfaces with a consistent type.	Electrics	Figure A.7
1.8	Convention	For terminals and bus couplers, the maximum number of connectible participants must not be violated.	Electrics	Figure A.8
1.9	Convention	Software variables must follow the Hungarian notation – e.g., *iName* for integer variables and *bName* for Boolean variables.	Software	Figure A.9
1.10	Convention	The call hierarchy of a Program must follow a tree structure.	Software	Figure A.10
1.11	Domain-specific	Global software variables with an assigned address (either input or output) must be read or written at least once.	Software	Figure A.11

Table 6.3. Inter-model inconsistencies investigated in evaluation stage 1

No.	Type	Description	Vocabulary	Reference
1.12	Correspondence	Each project documentation must be refined by a respective model in the *mechanics*, *electrics*, or *software* vocabularies.	Project, mechanics, electrics, and software	Figure A.12
1.13	Correspondence	Each mechanic part must be represented by an equivalent entry in the bill of material.	Mechanics and project	Figure A.13
1.14	Correspondence	Each electric automation hardware must be represented by an equivalent entry in the bill of material.	Electrics and project	Figure A.14
1.15	Correspondence	Each wired interface of a terminal should be represented by a corresponding global software variable – that is: variable type and address must match.	Electrics and software	Figure A.15

Results of inconsistency management

By means of the previously introduced inconsistency diagnosis and handling rules, the models can be analysed accordingly. Especially for the transition from scenario 12 in the PPU case study to scenario 13, an additional sensor *potentiometer* is added to the electrics model. By means of the specified inconsistencies, it can be identified, whether the potentiometer is wired correctly (inconsistency 1.7) to the respective bus terminals. Moreover, it can be analysed, whether appropriate entries have been added to the bill of material in the project model (inconsistency 1.14) – in case no entry is available, a new entry can be generated. Inconsistency 1.15 ensures that for the newly added sensor, a respective software variable is defined and linked. Hence, by means of the inconsistency rules, it can be ensured that the potentiometer is integrated correctly in the engineering models.

6.3.3. Synopsis

Through the basic inconsistencies used within this evaluation stage, it can be deducted that the presented inconsistency management approach is capable of specifying, diagnosing and handling simple inconsistencies in structural engineering models. However, the inconsistency diagnosis and handling rules used throughout evaluation stage 1 are of a basic nature – hence, more complex inconsistencies are focus of the evaluation stage 2.

6.4. Evaluation Stage 2: Evaluation of the Feasibility for an Industry-Style Application Example

Whereas the previous section illustrates the basic applicability of the proposed concept to engineering models in the automated production systems domain for the purpose of managing the basic inconsistencies introduced in Chapter 5, this section illustrates how additional background knowledge can be incorporated to draw conclusions on more complex inconsistencies. Therefore, additional models are incorporated for this section, which are introduced in Section 6.4.1 – namely background models that describe additional information on the configuration of modules in the automated production systems domain. The required inconsistency diagnosis and handling rules that are needed to draw the correct conclusions from the additional models are introduced in Section 6.4.2. The findings of this evaluation stage are summarized in Section 6.4.3.

6.4.1. Overview of the Incorporated Models

During engineering of automated production systems, besides the knowledge modelled explicitly in engineering models, a multitude of different background knowledge exists and must, hence, be taken into account. For one, such background knowledge incorporates, e.g., knowledge resulting from physical theories and laws. In addition, such knowledge may result from, e.g., manufacturers' device catalogues or heuristics that are not explicitly captured but implicitly available as configuration knowledge that influences, how the engineering solution of an automated production system looks like. Moreover, engineers use a multitude of different specification formalisms to not only capture the structure of an automated production system, but also its functional behaviour. Especially in the automated production systems domain, functional specifications such as adaptations of the Unified Modeling Language (UML) Timing Diagram are applied [Rös16].

As a consequence, this section investigates how the presented inconsistency management framework can be used to incorporate such additional knowledge. Therefore, two additional types of knowledge are incorporated in the following: First, configuration knowledge is used to identify, whether certain modules can be used under given environmental conditions or not. By means of this configuration knowledge, it is illustrated, how complex inconsistencies can be specified and diagnosed to support engineers in developing the overall engineering solution. Second, UML Timing Diagrams are investigated as one additional model type to capture functional specifications of engineering solutions.

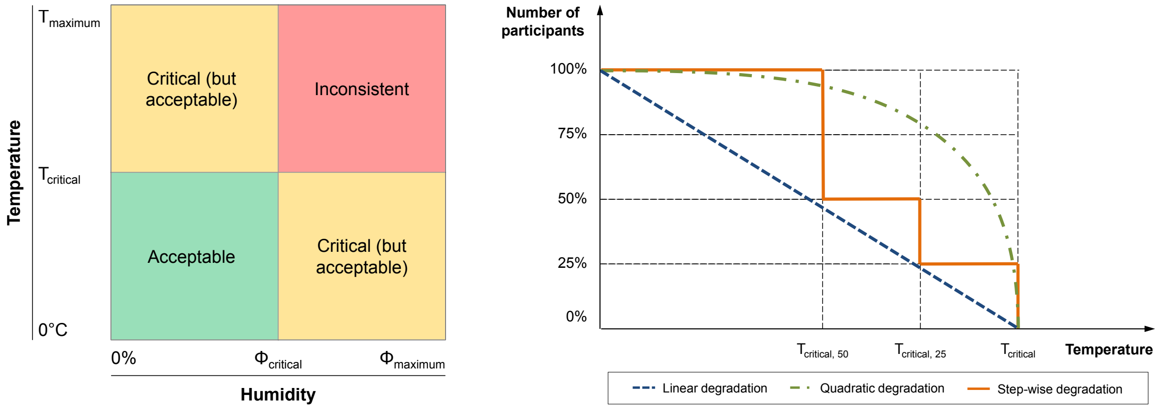
Capturing configuration knowledge: Module degradation

As argued beforehand, one essential source of knowledge to be incorporated during inconsistency management is configuration knowledge. Such configuration knowledge provides background information – often formulated in terms of heuristics – that is essential to ensure that an overall engineering solution is consistent. Within the context of this section, background knowledge on module degradation is considered. Specifically, it is assumed that, due to environmental conditions such as high temperature and humidity, modules degrade and, hence, cannot be used in these conditions.

In particular, the two cases illustrated in Figure 6.5 are considered in the following: A matrix-based criticality identification of device degradation due to humidity and temperature (see Figure 6.5a) and a function-based identification of coupler degradation due to temperature (see Figure 6.5b).

In a first case (Figure 6.5a), devices are classified according to a predefined critical ($T_{critical}$) and maximum temperature ($T_{maximum}$) as well as a predefined critical ($\phi_{critical}$) and maximum humidity ($\phi_{maximum}$). By means of a matrix-based classification of devices, it can be identified whether the environment conditions are acceptable ($T \leq T_{critical}$ and $\phi \leq \phi_{critical}$), critical but still acceptable ($T_{critical} < T \leq T_{maximum}$ or $\phi_{critical} < \phi \leq \phi_{maximum}$) or inconsistent ($T > T_{maximum}$ and $\phi > \phi_{maximum}$). For each device type, such critical and maximum parameters can be defined and used to identify, whether a certain device is acceptable under given environmental conditions or not.

The second case (Figure 6.5b) aims at identifying, which percentage of the maximum number of participants can be used for a bus coupler based on the current environment temperature. As with increasing temperature, the number of connectible participants decreases due to power degradation,



(a) Matrix-based identification of device degradation

(b) Function-based identification of coupler degradation due to temperature

Figure 6.5. Incorporating configuration knowledge to identify module degradation

this degradation needs to be captured explicitly for the purpose of inconsistency management. In the given case, three distinct types of degradations are possible: A linear degradation, in which the number of connectible participants decreases linearly, a cubic degradation, for which a cubic fall in number of connectible participants is observed, and a step-wise degradation, which aims at classifying the number of connectible participants in subsequent steps. Hence, using the respective parameters, the actual number of participants that can be connected to the coupler is determined based on the current temperature.

In order to capture such configuration knowledge explicitly, a dedicated model is required. Hence, a *degradation* vocabulary is introduced, which allows for modelling the respective information needed for inconsistency management (see Figure 6.6). In this vocabulary, a *DegradationModel* introduces respective *DeviceDegradations*, in which such degradation parameters are bound to manufacturer-specific *orderNumber*. Hence, each *DeviceDegradation* is applicable for any device, which is identified according to the *orderNumber*. In order to capture the information necessary for the matrix-based classification of the device degradation according to temperature and humidity, both a *HumidityValue* and a *TemperatureValue* can be assigned to the degradation with their accoding *criticalValues* and *maximumValues*. Additionally, if a *CouplerDegradation* is applied for the respective *orderNumber*, the modeller can assign a *LinearDegradation*, a *CubicDegradation*, or a *StepWiseDegradation* to denote the module's behaviour under high temperature.

In an exemplary modelling set-up, 4 different device types with their according order number are considered: three Profinet couplers (order numbers A-01-123, B-01-123, and C-01-123) as well as a micro switch (order number D-01-123). As indicated in Table 6.4, different values are given for the respective degradation parameters and different types of coupler degradations are defined for the respective device types. By means of this information, all necessary input is defined for the purpose of identifying, whether a device can be use under given environment conditions or not.

It should be noted that the illustrated cases are used for illustrative purposes and do not reflect real-world behaviour. Rather, these examples serve the purpose of illustrating, how more complex information can be incorporated within the inconsistency management framework.

Capturing functional specifications: UML Timing Diagrams

In addition to the configuration knowledge introduced previously, functional specifications are often used in order to specify the system's intended behaviour. One of such functional specifications is the UML Timing Diagram, which is often used for domain-specific adaptations in the automated production systems domain [Rös16].

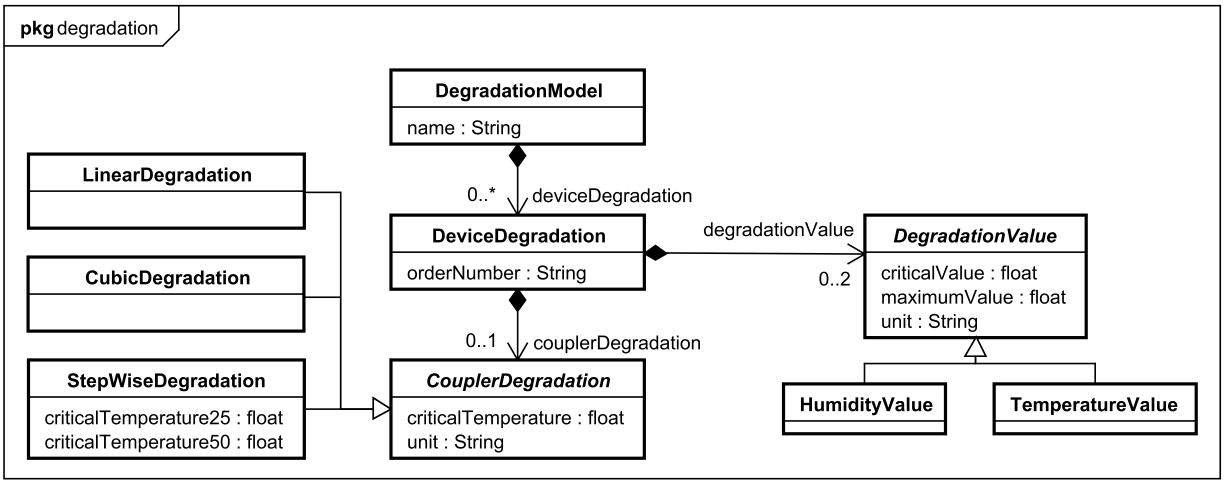


Figure 6.6. Vocabulary to describe device degradation parameters

Table 6.4. Exemplary degradation model for devices and couplers in a tabular notation

Order no.	Description	T_{crit}	T_{max}	ϕ_{crit}	ϕ_{max}	Coupler degradation
A-01-123	Profinet coupler	40°C	55°C	90%	95%	Linear degradation (T_{crit} = 40°C)
B-01-123	Profinet coupler	35°C	45°C	80%	90%	Cubic degradation (T_{crit} = 35°C)
C-01-123	Profinet coupler	35°C	45°C	80%	90%	Step-wise degradation (T_{crit} = 35°C, $T_{crit,25}$ = 30°C), $T_{crit,50}$ = 25°C
D-01-123	Micro switch	50°C	60°C	95%	100%	n/a

An example for such a Timing Diagram is illustrated in Figure 6.7. Therein, two distinct so-called *Interactions* are defined: One for the demand that the crane stops at end position 1 when turning clockwise and one for the demand that the crane stops at end position 2 when turning counter-clockwise. As can be seen from the Figure, *Lifelines* are used to specify the different variables' *StateInvariants*. For instance, the Lifelines *bCraneAtEnd1* and *bCraneAtEnd2* correspond to the respective software variables in the control program. By means of the specifications in the Timing Diagram, the intended behaviour of these variables is defined. Therefore, it must be ensured that each Lifeline in a Timing Diagram corresponds to an appropriate software variable.

6.4.2. Diagnosing and Handling the Inconsistencies

To diagnose the previously introduced potential inconsistencies, a couple of inconsistency diagnosis rules must be formulated (see Table 6.5). These inconsistencies are captured by means of 3 inconsistency rules. For simplicity reasons, only a rough description of the different inconsistencies is illustrated in the following – a detailed overview of all inconsistencies can be found in Appendix A.

A first rule is used to identify, whether any device (identified through its order number) is used although its specified critical humidity or temperature are exceeded under the given environment conditions (see inconsistency 2.1 in Table 6.5). Hence, by means of the matrix-based device classification introduced in Figure 6.5a, it can be identified, whether any device should be replaced or not. A second inconsistency rule identifies, whether the number of connections to a bus coupler meets the allowed number of connections considering power degradation (see inconsistency 2.2). By means

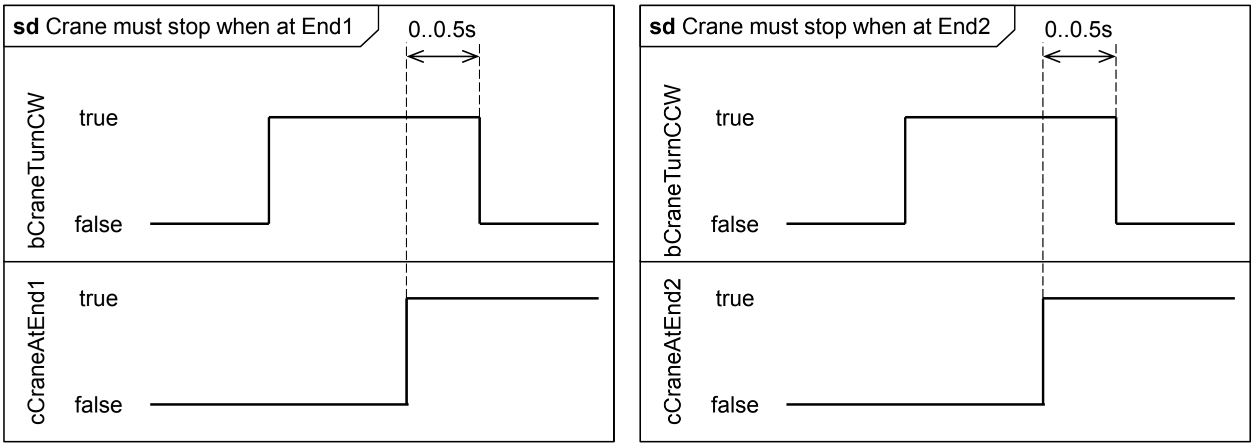


Figure 6.7. UML Timing Diagrams as the basis to specify module behaviour

of this rule, the different degradation patterns introduced in Figure 6.5 can be identified. Finally, a third rule identifies, whether all lifelines in a UML Timing Diagram correspond to respective variables in the *software* vocabulary (see inconsistency 2.3). Hence, it is ensured that the functional specification is consistent to the respective *software* model in the engineering project as introduced in Figure 6.7.

Table 6.5. Inconsistencies investigated in evaluation stage 2

No.	Type	Description	Vocabulary	Reference
2.1	Correspondence	Devices (identified through their order number) should not be used if critical temperature or critical humidity are exceeded under current environment conditions.	Project and degradation	Figure A.16
2.2	Correspondence	Bus couplers' (identified through their order number) number of connections must match the degradation pattern (linear, cubic, or step-wise degradation).	Electrics and degradation	Figures A.17 to A.19
2.3	Correspondence	Each software variable should be equivalent to a respective UML lifeline (that is: a link must exist and the types of both software variable and UML lifeline must match).	Electrics and UML	Figure A.20

As a consequence, by means of the introduced inconsistencies, more complex knowledge can be incorporated during inconsistency management. For instance, depending on the type of Profinet bus coupler used within the PPU models, the number of connectible bus terminals varies under the given environment conditions. Especially as adding the *potentiometer* to the models requires an additional analogue input terminal, it must be decided whether the Profinet coupler is sufficient or whether an additional coupler is needed for the purpose of integrating the potentiometer (inconsistency 2.2).

Moreover, by means of analysing, whether the potentiometer's software variable is used within the Timing Diagram or not, it can be ensured whether or not a functional specification is made for the potentiometer's behaviour (inconsistency 2.3).

6.4.3. Synopsis

The inconsistencies defined in this evaluation stage verify that also complex background knowledge can be incorporated in the presented inconsistency management framework – e.g., configuration knowledge that uses information on environment conditions to analyse whether or not a system's configuration is consistent. Moreover, through establishing links to behaviour models such as Timing Diagrams, inconsistencies between structure models and behaviour models can be specified, diagnosed and handled. However, the presented inconsistency management framework is limited to structural inconsistencies – that is: to inconsistencies that can be drawn from a structure or behaviour model without actually executing the model.

6.5. Evaluation Stage 3: Comparison with Another Inconsistency Management Approach

Following the evaluation of the basic applicability of the proposed inconsistency management framework in Sections 6.3 and 6.4, this evaluation stage compares the proposed framework with the one proposed in [Fel+16c], which makes use of solely Model-Driven Engineering (MDE) technologies within the EMF [Ecl16c] as well as the Eclipse Epsilon Framework [Ecl16b]. As a basis to perform the evaluation, the models and inconsistencies described in [Fel+15a; Fel+16c] are taken into account and, based on these models and inconsistencies, the feasibility and performance of both approaches while managing these inconsistencies are compared.

In the following, the investigated engineering models are introduced in Section 6.5.1 with their according mediation rules (Section 6.5.2). Subsequently, the distinct types of inconsistencies that are considered as a basis to compare both approaches are described in Section 6.5.3. Based on these findings, the approaches are compared in Section 6.5.4 – the results of this evaluation stage are summarized in Section 6.5.5.

6.5.1. Overview of the Incorporated Models

As a basis to compare the two inconsistency management frameworks, three distinct engineering models are taken into account, which can be mapped to distinct categories of modelling languages:

- A *planning modelling language* uses the *Function-Behaviour-Structure* modelling approach and was developed by Kammerl et al. [Kam+15] to select upon structural modules (*Structure*) based on the functional requirements on the system (*Function*) and the intended behaviour that can be derived from these requirements (*Behaviour*).
- The *Systems Modeling Language (SysML)* is used to detail the structure from the *planning model* and provides a detailed, logical architecture of the system under investigation within a *system model.*
- Within a *MATLAB/Simulink* simulation model, it is aimed at verifying, whether the system's properties defined in the *system model* can satisfy the requirements demanded in the *planning model.*

A detailed overview of the models under investigation can be found in Figure 6.8 as well as in [Fel+15a]. As can be seen from the Figure, there are certain links in between these different models. Therein, *RefinesLinks* are expected between the *Modules* being selected in the *planning model* as well as the *Blocks* defined in the *system model* (see ① in Figure 6.8). Accordingly, *EquivalentToLinks* (see ②) are expected between properties defined in the *system model* as well as *Constants* in the *simulation model*. Finally, *SatisfiesLinks* refer to *Displays* in the *simulation model* that must satisfy the *Requirements* that have been defined in the *planning model* (see ③).

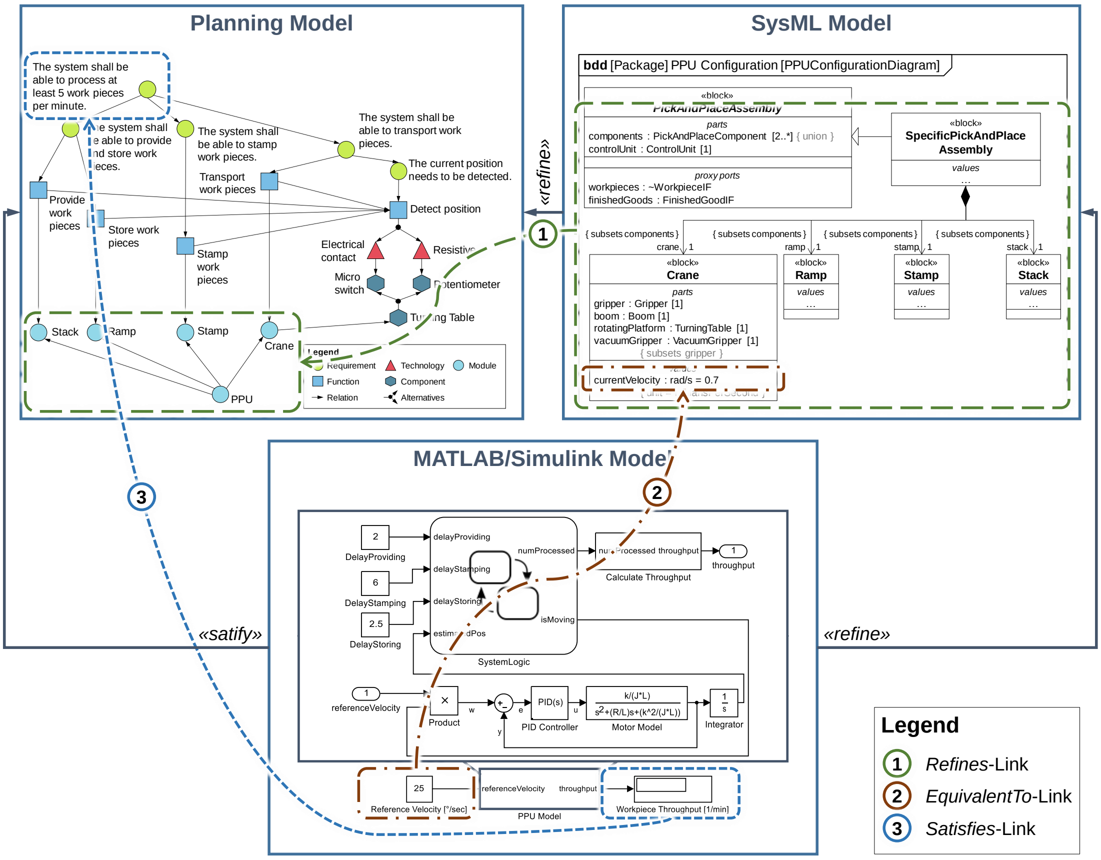


Figure 6.8. Models for the comparison (from [Fel+15a; Fel+16c])

Certainly, these links must be regarded from a metamodel view as they, from an inconsistency management perspective, must be described for the different metamodel elements. Consequently, Figure 6.9 shows the different metamodels for these engineering models as well as the links that are expected to exist between them.

6.5.2. Mediation Between the Resulting Vocabularies

As discussed in Chapter 5, crucial to an efficient inconsistency management is an abstraction mechanism that reduces the number of necessary inconsistency diagnosis and handling rules to be defined. Consequently, the respective modelling languages introduced for the comparison are mediated with the respective *common* vocabulary as illustrated in Figure 6.10. Therein, as can be seen from the Figure, all three modelling languages – that is, the *planning*, the *SysML*, and the *MATLAB/Simulink* modelling languages – are mediated with the *common* vocabulary. Accordingly, the *Quantity, Unit, Dimension and Type (QUDT)* vocabulary is being used to define the respective units being used throughout the modelling example.

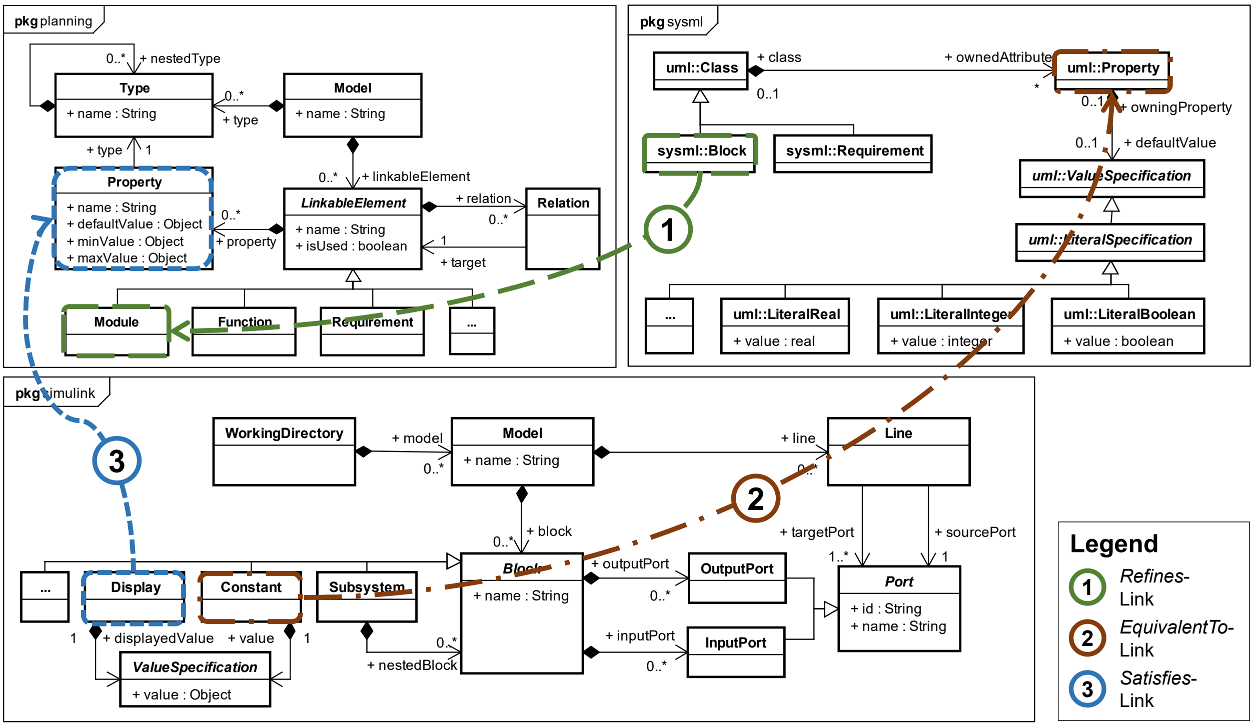


Figure 6.9. Metamodels for the comparison (from [Fel+15a; Fel+16c])

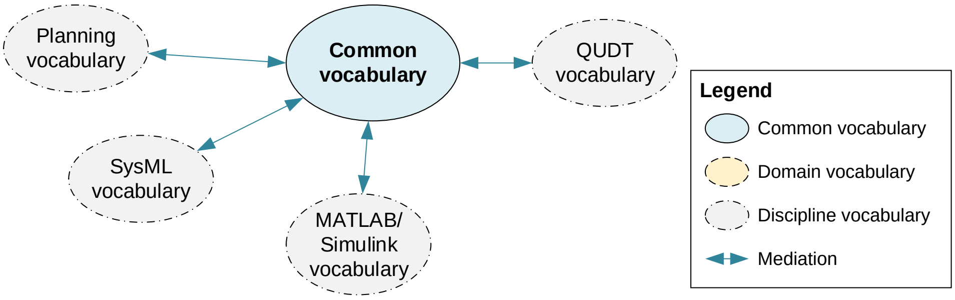


Figure 6.10. Semantic mediation between vocabularies that are focused on in the comparison

6.5.3. Diagnosing and Handling the Inconsistencies

For the comparison of both approaches, a set of intra-model and inter-model inconsistencies is regarded. Certainly, these inconsistencies do not cover all possible inconsistencies that can exist in heterogeneous engineering models, but, however, these inconsistencies are regarded as sufficient for the purpose of comparing both approaches as they stem from all inconsistency categories relevant to inconsistency management – namely notational, conventional, correspondence and domain-specific inconsistencies. The different inconsistencies considered for the comparison are introduced in the following.

Intra-model inconsistencies

In order to compare both approaches regarding their capability to identify and handle intra-model inconsistencies, 4 distinct inconsistencies are defined (see Table 6.6). Within a first inconsistency (see inconsistency 3.1 in Table 6.6), it is demanded that all entities that are mediated to *Elements* in the *common* vocabulary follow the *UpperCamelCase* naming convention – that is, the inconsistency defines a particular regular expression to be followed by all candidate elements. Accordingly, a second inconsistency (see inconsistency 3.2) demands that all entities mediated to *Properties* in the *common* vocabulary follow the *lowerCamelCase* naming convention. Further on, a domain-specific inconsistency 3.3 demands that all mass properties in the entire set of models are greater than or equal to zero. Finally, a fourth inconsistency 3.4 demands a maximum hierarchy level – that is, *Elements* in the *common* vocabulary must not have 4 or more transitive *contains* relations.

Table 6.6. Intra-model inconsistencies investigated in evaluation stage 3

No.	Type	Description	Vocabulary	Reference
3.1	Convention	Each element in any vocabulary must be named in *UpperCamelCase* style.	Common	Figure A.21
3.2	Convention	Each property in any vocabulary must be named in *lowerCamelCase* style.	Common	Figure A.22
3.3	Domain-specific	Each mass property in any vocabulary must not be negative.	Common	Figure A.23
3.4	Convention	Each element in any vocabulary must not have more than 3 transitive *contains* relations – hence, a maximum hierarchy level of 3 must not be violated.	Common	Figure A.24

Inter-model inconsistencies

Accordingly, to compare both approaches regarding their capability to identify and handle inter-model inconsistencies, 4 inconsistencies are defined (see Table 6.7). Inconsistencies 3.5 and 3.6 in Table 6.7 identify, whether all *Modules* in the *planning* vocabulary are refined by respective *Blocks* in the *sysml* vocabulary. Therein, it is checked whether all of these linked *Blocks* have identical names a the respective *Modules*. Moreover, it is ensured that the hierarchy of *Modules* is equivalent to the hierarchy of *Blocks* (i.e., whether for each *relation* in the *planning* vocabulary, a respective *ownedAttribute* can be identified). Moreover, respective inconsistencies to identify, whether all top-level *simulink Displays* (inconsistency 3.7) and *Constants* (inconsistency 3.8) are linked appropriately are defined. For these inconsistencies, it is checked whether all of these entities have values that are consistent to each other – for the sake of simplicity, the mediation to the *qudt* vocabulary is not illustrated in the respective figures.

Results of inconsistency management

By means of the previously introduced inconsistency diagnosis and handling rules, the models can be analysed accordingly. When adding the *potentiometer* to the engineering models, it can be ensured that the additional module is not only added to the *planning* model, but also refined by the respective *SysML* model (inconsistencies 3.5 and 3.6). Hence, it can be ensured that the potentiometer is integrated appropriately in the engineering models.

6.5.4. Comparison of the Two Approaches

As shown by the inconsistency diagnosis and handling rules introduced in this evaluation stage, further engineering models can be investigated by the presented inconsistency management frame-

Table 6.7. Inter-model inconsistencies investigated in evaluation stage 3

No.	Type	Description	Vocabulary	Reference
3.5	Correspondence	Each *planning* Module should be refined by an equivalently named *SysML* Block.	Planning and SysML	Figure A.25
3.6	Correspondence	Each *planning* Module hierarchy should be refined by an equivalent *SysML* hierarchy – that is: each *Module* with a child *Module* should have a corresponding *Block* with a child *Block*.	Planning and SysML	Figure A.26
3.7	Correspondence	Each top-level *simulink* Display should satisfy a *planning* Property – that is: the Display's *displayedValue* must be in the range of the Property's *minValue*, *maxValue*, and *defaultValue*.	Simulink and planning	Figure A.27
3.8	Correspondence	Each top-level *simulink* Constant should be equivalent to a respective *SysML* Property – that is: the Constant's *value* must be equivalent to the Property's *defaultValue*.	Simulink and SysML	Figure A.28

work. An alternative implementation of an inconsistency management framework that makes use of the Eclipse Epsilon Framework [Ecl16b], in particular of the Epsilon Validation Language (EVL), is presented in [Fel+15a; Fel+16c]. Although both frameworks – the inconsistency management framework presented in this dissertation and the one presented in [Fel+15a; Fel+16c; Fel+19] – are capable of managing the inconsistencies formulated in this evaluation stage, there are essential differences in their operationalization.

For one, the use of mediation techniques allows for abstracting the complexity for managing intra- and inter-model inconsistencies. For instance, inconsistencies in several model types can be managed by specifying exactly one inconsistency diagnosis and handling rule on the common vocabulary (see naming inconsistency 3.1). In contrast, approaches such as the EVL require to specify one inconsistency diagnosis and handling rule per model type. In the presented example in this evaluation stage, diagnosing and handling a naming convention for all entities in all models would require 3 disparate diagnosis and handling rules. Analogously, mediation of, e.g., physical units to the QUDT vocabulary allow for analysing type and value consistency of properties on a more general level. Hence, it can be argued that by means of mediating the distinct engineering models, the effort in creating and maintaining rules is decreased. This is especially important for integrating such an approach for industry settings, in which a high degree of flexibility is required.

However, whereas approaches such as the EVL rely on well-established standards and tools in MDE, especially as EVL is based on the well-known Object Constraint Language (OCL), the use of Semantic Web Technologies such as RDF/RDFS and SPARQL is relatively new to MDE. As a consequence, two main factors can be expected when applying the different frameworks to real-world scenarios: (1) experts in the field of MDE are not yet familiar with Semantic Web Technologies and (2) integrated tools such as the Eclipse Epsilon Framework are not yet available to leverage Semantic Web Technologies for MDE and must be created. Especially the additional need to transform models from modelling tools such as the EMF to Semantic Web-based technologies must be overcome through appropriate integrated standards and tools.

One means to overcome this challenge and to further increase operationalizability is to provide graphical means to model inconsistencies as suggested in [Fel+19]. As could be shown in by means of experiments conducted together with different experts in the field of MDE, graphically modelled inconsistency patterns can help in specifying as well as maintaining different inconsistency types. Hence, this type of modelling can help engineers in managing inconsistencies.

In summary, it can be argued that both an inconsistency management framework based on Semantic Web Technologies such as the one presented in this dissertation and frameworks based on solely MDE technologies such as the one presented in [Fel+15a; Fel+16c; Fel+19] provide valuable benefits. It is therefore essential to bring the best of both together in order to create appropriate inconsistency management support for engineers in the automated production systems domain.

6.5.5. Synopsis

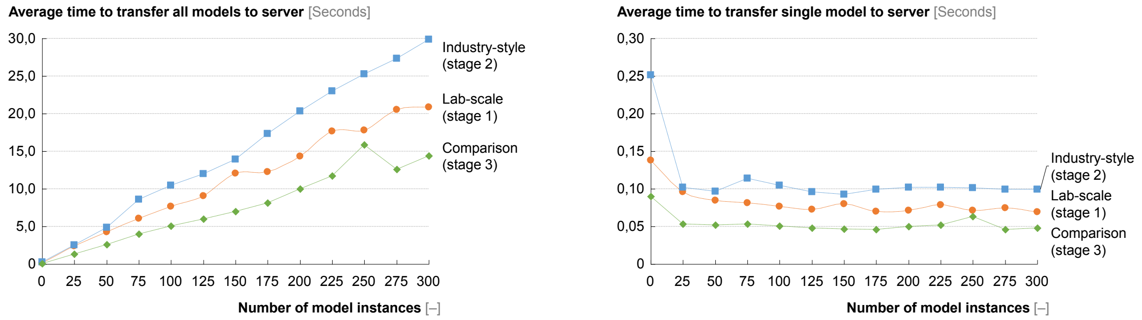
The model and inconsistency types used throughout this evaluation stage illustrate that the presented inconsistency management framework is able to incorporate distinct modelling languages such as domain-specific languages, standard modelling languages and tool-specific languages. Moreover, the comparison with a framework based on the EVL revealed that several benefits (e.g., decreased complexity through mediation techniques) and limitations (e.g., experts not yet familiar with Semantic Web Technologies) can be observed. It is, hence, inevitable to benefit from the best of both in order to create appropriate inconsistency management approaches for real-world scenarios.

6.6. Assessment of the Performance and Scalability

The practical feasibility of the proposed inconsistency management framework strongly depends on its computational performance for industry-scale systems. Such systems often contain thousands of entities and relations. In fact, the engineers, who are using the proposed inconsistency management approach, expect feedback within an adequate response time. Consequently, the aim of the assessment of the framework's performance and scalability is to estimate, how the approach performs for industry-scale model.

In order to identify the performance and scalability of the inconsistency management framework, two distinct experiments are performed for the three evaluation stages introduced beforehand. Both experiments are performed on a standard office PC (Intel(R) Core(TM) i7-6600U CPU @ 2.60 GHz 2.70 GHz, with 16 GB of physical memory on a Windows 8.1 Enterprise 64 bit operating system) with the Java Virtual Machine in its version 1.8.0. As the basis for performing the experiments, the prototypical software implementation introduced in Section 6.1 is used, in particular by means of the Eclipse Neon.3 Release 4.6.3 for creating and maintaining the involved models as well as the Apache Jena Fuseki Release 2.6.0, which serves as the SPARQL endpoint.

In a first experiment (see Figure 6.11), the time to transfer the modelled information to the inconsistency management server is measured. Therein, the transfer time is measured for different numbers of instances of the involved models (from 0 to 300 model instances). For each number of model instances, 10 transfer runs were performed and the average of all runs was used for comparison. As can be seen from the results in Figure 6.11a, the transfer time increases linearly with an increasing number of model instances. As the models of the evaluation stage 2 (Section 6.4) are more complex than the ones of stages 1 (Section 6.3) and 3 (Section 6.5), the highest transfer time can be identified for the models in evaluation stage 2. However, with a transfer time of approximately 29.9 seconds for 300 model instances, it can be argued that the model transfer scales appropriately also for complex models. In addition, it can be argued that transferring all model instances at the same time is not a primary use case. Rather, engineers incrementally change and extend models – hence, model fragments will be transferred to the server for inconsistency management purposes step by step. As can be seen from Figure 6.11b, the time to transfer a single model to the server is nearly independent from the number of model instances that is currently captured in the server. Therefore, independent from the size of the entire model, model fragments can be added to the server in short transfer times.



(a) Average time to transfer all models **(b)** Average time to transfer a single model

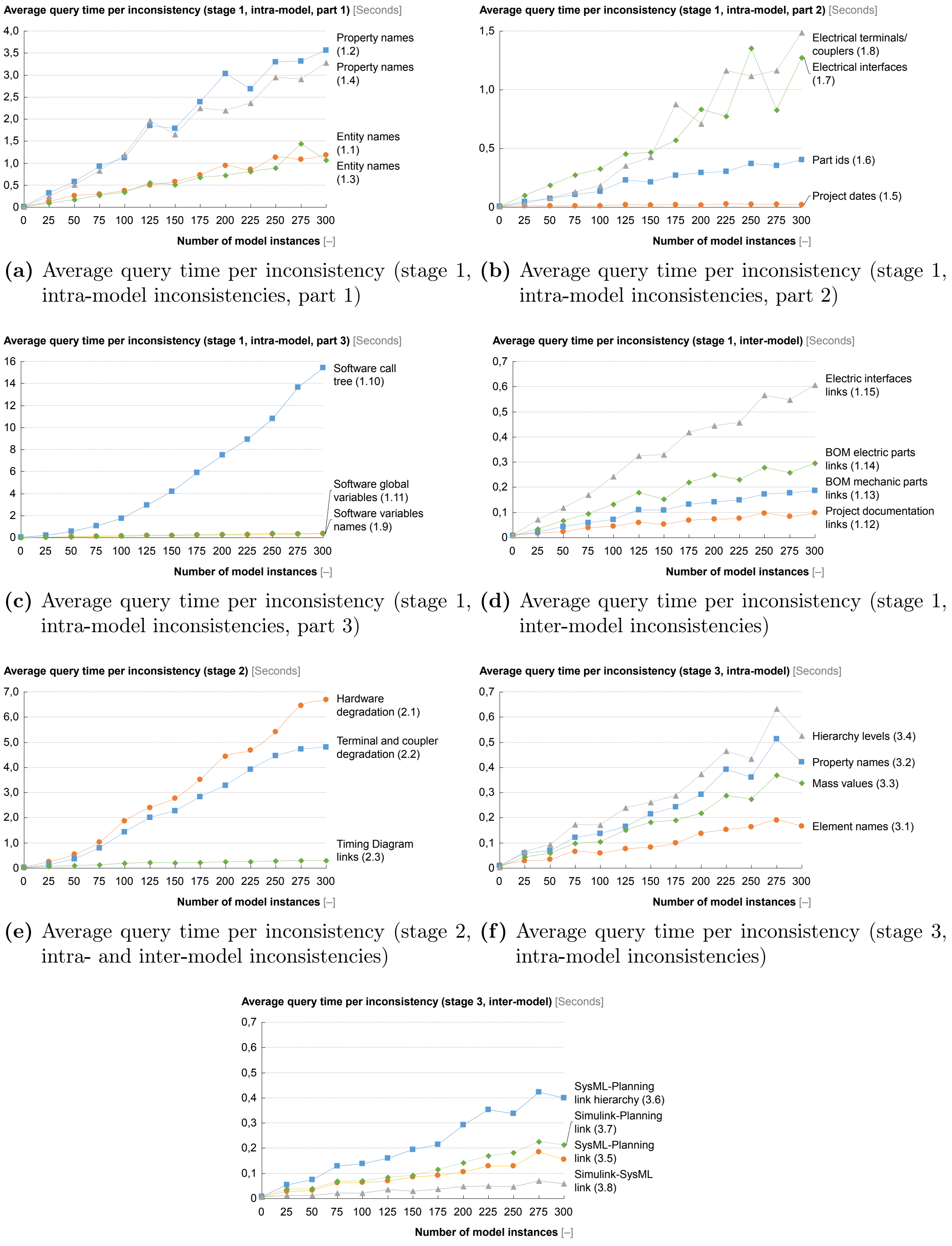
Figure 6.11. Results of the transfer time analysis

A second experiment (see Figure 6.12) analyses the query times for the different inconsistencies and evaluation stages. In particular, the query time is measured for different numbers of instances of the involved models (from 0 to 300 model instances). For each number of model instances and inconsistency, 10 query runs were performed and the average of all runs was used for comparison purposes. As can be seen from the results, the query time for the different inconsistencies strongly varies with the complexity of the respective inconsistency and, at the same time, with the complexity of the involved types of models. For instance, the analysis for the models of evaluation stage 1 illustrate that inconsistencies, which depend on mediations to other vocabularies such as naming inconsistencies that are analysed in the common or mechatronics vocabulary (see Figure 6.12a, approximately 3.6 seconds to analyse property naming conventions in 300 model instances in inconsistency 1.2) require longer query executions than inconsistencies that are analysed in a specific discipline (see Figure 6.12b, approximately 0.3 seconds to analyse software variable naming conventions in 300 model instances in inconsistency 1.9). This is especially due to the fact that inconsistencies depending on mediations require mediation rules to be performed – inconsistencies in a specific discipline do not require the execution of mediation rules. Moreover, the analyses reveal that complex inconsistency definitions (e.g., inconsistencies 1.7, 1.8 and 1.15 require count operations to, e.g., identify the number of currently wired hardware elements) also require longer execution times. A high complexity of the inconsistency query can be observed for the call tree inconsistency (inconsistency 1.10 in Figure 6.12c), which requires to iterate over all call dependency in the respective software dependency model for each individual call dependency. Further examples of such complex inconsistency definitions are the hardware degradation inconsistency in stage 2 (inconsistency 2.1 in Figure 6.12e) as well as the hierarchy level inconsistency in stage 3 (inconsistency 3.4 in Figure 6.12f).

All in all, it can be argued that appropriate query times for diagnosing inconsistencies can be achieved. Further on, instead of providing feedback on inconsistency diagnosis results instantly, the inconsistency management approach could also be integrated, e.g., in continuous integration processes. In such cases, the actual time to query for an inconsistency becomes less important, as inconsistency diagnosis can be performed, e.g., over night.

6.7. Summary

Summarizing the evaluation of the presented inconsistency management framework, it can be deduced that both simple and complex inconsistency diagnosis and handling rules can be specified



(a) Average query time per inconsistency (stage 1, intra-model inconsistencies, part 1)

(b) Average query time per inconsistency (stage 1, intra-model inconsistencies, part 2)

(c) Average query time per inconsistency (stage 1, intra-model inconsistencies, part 3)

(d) Average query time per inconsistency (stage 1, inter-model inconsistencies)

(e) Average query time per inconsistency (stage 2, intra- and inter-model inconsistencies)

(f) Average query time per inconsistency (stage 3, intra-model inconsistencies)

(g) Average query time per inconsistency (stage 3, inter-model inconsistencies)

Figure 6.12. Results of the query time analysis

and executed by means of the approach. By comparing the presented framework with an alternative implementation, several benefits and limitations can be derived. A basis performance and scalability evaluation reveals that the presented inconsistency management framework is appropriately scalable, also for more complex real-world scenarios. A detailed discussion of the evaluation results with regard to the requirements introduced in Chapter 3 can be found in the next chapter.

Chapter 7.

Discussion of the Results

Based on the results of the evaluation (Chapter 6), this chapter is devoted to discussing the strengths and limitations of the presented framework. First, it is assessed, whether the concept addresses the requirements that are introduced in Chapter 3 to a sufficient degree (Section 7.1). Second, the strengths (Section 7.2) and limitations (Section 7.3) are elaborated in detail.

7.1. Assessing the Fulfilment of the Requirements

Based on the evaluation results (Chapter 6), this section aims at assessing, whether the proposed inconsistency management framework addresses the requirements introduced in Chapter 3.

As can be seen from both application examples (Sections 6.3 and 6.4), the inconsistency management framework allows for representing the different engineering models within Resource Description Framework (RDF), thereby providing a common, syntactical formalism for all of the involved models (see Requirement 1.1). Accordingly, by means of the mediation mechanism, the discipline-specific entities are transformed to more abstract concepts that are common to multiple disciplines (see Requirement 1.2). In addition, respective link models are created to explicitly define the links in between the different entities in the models (see Requirement 1.3). As a consequence, Requirement 1 “Model knowledge base” is evaluated to be fully fulfilled.

For evaluating, whether the proposed inconsistency management framework is capable of diagnosing and handling the typical types of inconsistencies (Chapter 2) – namely, notational, conventional, correspondence and domain-specific inconsistency rules – representative samples are introduced for both application examples (Sections 6.3 and 6.4). As indicated for both, all types of inconsistencies are specified by means of the SPARQL Protocol and RDF Query Language (SPARQL) Query Language (see Requirement 2.1), and, when executed against the model knowledge base, are identified, located and classified as expected (see Requirement 2.2). Accordingly, handling rules can be specified (see Requirement 3.1) and used (see Requirement 3.2) by means of the SPARQL Update Language. Therefore, Requirement 2 “Inconsistency diagnosis” and Requirement 3 “Inconsistency handling” are also evaluated to be fulfilled.

According to Requirement 4, it is essential to provide the means to measure diagnosed inconsistencies (see Requirement 4.1) and to assess, what the impact of a handling action is (see Requirement 4.2). By means of the metadata (e.g., severity of an inconsistency, role of an involved entity), stakeholders are provided with the basis to identify the importance of a particular inconsistency. Accordingly, as handling actions are presented to stakeholders with the related entities that are involved while handling the inconsistency, the responsible stakeholder can decide upon the handling action to be taken. However, sophisticated impact analysis models are out of scope of this dissertation and must be identified for the specific context in which the inconsistency management framework is employed. Therefore, Requirement 4 “Measurement and assessment capabilities” is evaluated to be fulfilled partly.

As discussed in Chapter 3, essential to the success of an inconsistency management approach is its support to be operationalized in industrial settings. For one, as the proposed framework incorporates technologies from the Semantic Web initiative, which are standardized by the World

Wide Web Consortium (W3C), its extensibility, e.g., to incorporate additional types of models or inconsistencies within company- or project-specific settings, is enabled (see Requirement 5.1). However, regarding its comprehensibility (see Requirement 5.2), especially with regard to experts in the automated production systems domain, which are mostly not familiar with such technologies, the proposed concept has its limitations – in particular in comparison to techniques from the Model-Based Systems Engineering (MBSE) domain (Section 6.5), which experience better acceptance. Nevertheless, when analysing the proposed framework's performance and scalability, it can be concluded that also large-scale models can be managed (see Requirement 5.3). Consequently, Requirement 5 "Operationalization" is identified to be fulfilled mostly.

7.2. Strengths of the Proposed Framework

In this section, the strengths that can be derived for the proposed inconsistency management framework are discussed.

As discussed in Chapter 4, inconsistency management approaches can be broadly divided into logical reasoning and theorem proving (Section 4.2.1), rule- and pattern-based (Section 4.2.2) as well as model synchronization approaches (Section 4.2.3). The proposed inconsistency management framework can be assigned to the group of rule-based approaches, which makes use of inconsistency patterns that describe the conditions for the existence of an inconsistency. Accordingly, the proposed framework allows for flexibly extending and reducing the knowledge base – and, hence, adoptions for project- or company-specific purposes can be implemented easily. In addition, as the framework makes use of Semantic Web Technologies (namely RDF and SPARQL), which are standardized by the W3C, additional model types and disciplines can be integrated easily. Consequently, by means of the proposed framework, an extensible, yet efficient concept for inconsistency management approach is enabled.

Especially when comparing the framework to other common approaches, the benefits can be obtained: Instead of synchronizing models in a bi- or uni-directional way (cf. Section 4.2.3), the proposed framework aims at linking the models at the points, where inconsistencies are believed to occur frequently. Hence, instead of using an integrated modelling language or formats in terms of a "world model" for the automated production systems domain (Section 4.1.1), which implies the application of a single modelling language or format for the entire engineering process, existing models can be integrated for the purpose of managing inconsistencies within or in between them – as long as they are represented in RDF. Accordingly, semantic mediation techniques can be put in place to decrease the specification effort, as the amount of necessary inconsistency (diagnosis and handling) rules is lowered.

7.3. Limitations of the Proposed Framework

Although a huge step towards managing inconsistencies within and in between heterogeneous engineering models of automated production systems is made with the proposed framework, it is subject to several limitations, which are discussed in this section.

One essential limitation of the proposed framework is that all models to be considered for inconsistency management must have a precise syntax and semantics – that is: they must comply to the Object Management Group (OMG)'s Meta Object Facility (MOF) [OMG15a]. Although this assumption is valid for a multitude of common modelling languages such as Unified Modeling Language (UML) and Systems Modeling Language (SysML) as well as for most Domain-specific Languages (DSLs) in the field of MBSE, there certainly exists a multitude of additional model types that cannot be addressed by the proposed framework. For one, non-technical models such as the ones from socio-technical or psychological disciplines often comprise theories or statistical data.

These models are essential when incorporating a holistic perspective on the engineering process, thereby also incorporating aspects of, e.g., how engineering teams interact with each other. In addition to models, engineering documents are often used within the automated production systems domain – especially if technicians or non-trained personnel are involved, which makes use of, e.g., instruction lists for commissioning or maintenance tasks. These types of models and documents, which are often represented in a (textual or graphical) unstructured format, are not incorporated within the framework presented in this dissertation. First efforts towards integrating, e.g., textual documents with manual annotations are proposed in [Kol$^+$17] – these efforts must be integrated into inconsistency management frameworks in future research.

A further aspect that is not considered within this dissertation is the additional knowledge that is needed to employ the inconsistency management framework. Especially with regard to the Semantic Web Technologies, which are used as the basic technology throughout the entire framework, it is obvious that additional experts are required to employ such an inconsistency management framework in industrial practice. Consequently, it is essential to find abstraction mechanisms that allow non-experts in this field to efficiently and effectively specify, diagnose and handle inconsistencies. One means to overcome this challenge is to provide a visual specification formalism for inconsistency diagnosis and handling rules – e.g., by means of the triple graph pattern-based specification language presented in [Gue$^+$13] and used in [Fel$^+$16c; Fel$^+$19] for a first, interactive inconsistency management approach. Hence, inconsistency diagnosis and handling rules are defined through visual modules – stakeholders do not need to understand the formalism behind these models. A first study that has been performed together with experts in the Model-Driven Engineering (MDE) domain showed that such graphically modelled patterns can simplify the inconsistency management process as well as increase efficiency [Fel$^+$19]. In addition to the specification of inconsistency rules, the amount of inconsistencies that are diagnosed and, hence, need to be handled in large-scale models can be tremendous – especially as industrial systems consist of thousands of entities. Consequently, appropriate visualization techniques such as the one proposed in [Bas$^+$15] are needed to condense the information to the necessary amount. First efforts towards integrating inconsistency management within visualization techniques were made in [Fel$^+$17]; however, their applicability for real-world use cases must be verified in the future.

Furthermore, inconsistencies in the proposed framework are specified by means of graph patterns (i.e., SPARQL queries). One limiting factor is, hence, the type of inconsistencies that can be managed. For one, inconsistencies that are subject to uncertainties, which is often the case for physical automated production systems, are currently investigated in the framework. Additional mechanisms such as the ones proposed in [Her15] can, due to the extensibility of standards such as RDF, easily be integrated for this purpose. In addition, this dissertation focuses on static data – i.e., although able to incorporate both structure and behavioural models, the dynamics of an automated production system cannot be analysed. As a consequence, expansions by statistical simulation and formal verification techniques (e.g., [Bec$^+$15]) are necessary to address this purpose.

Finally, one essential question to be answered in future research must address the source for the distinct inconsistency rules to be maintained during inconsistency management. Although companies in the automated production systems domain more and more start to specify the rules they need to obey during engineering (e.g., regarding coding conventions [PLC16]), the essential sources for inconsistency diagnosis and handling rules must be identified. Although systems engineering is gaining more and more importance in the automated production systems domain [KV13] and systems engineers become candidates for the role of inconsistency managers, such roles must be defined and established in the future to ensure an integrated inconsistency management process throughout the entire engineering of automated production systems.

Chapter 8.

Conclusions and Outlook

The ever-increasing complexity of automated production systems arises the need to provide appropriate methods to improve the efficiency and effectiveness of the engineering. To overcome this challenge, Model-Based Systems Engineering (MBSE) provides a suitable way by means of models, which abstract the reality, thereby focusing on a certain point of interest. However, the multitude of disciplines involved during engineering leads to manifold, heterogeneous models that are created and maintained throughout this process. As these models overlap – that is, they refer common aspects of the system under investigation – it is likely that inconsistencies within or in between these models occur. Especially for automated production systems engineering, these inconsistencies can cause severe consequences if they are not diagnosed and handled in a proper way.

This dissertation analyses the requirements to be fulfilled by an inconsistency management approach that allows for diagnosing and handling inconsistencies within and in between heterogeneous engineering models of automated production systems (Chapter 3). Based on these findings and a comparison of the related work (Chapter 4), a knowledge-based inconsistency management framework is suggested (Chapter 5), which makes use of Semantic Web Technologies – in particular the Resource Description Framework (RDF) and the SPARQL Protocol and RDF Query Language (SPARQL) – for the purpose of specifying, diagnosing and handling inconsistencies in a (semi-)automatic manner. Besides typical engineering models, e.g., from the mechanical, electrical and software engineering disciplines, the framework is capable of incorporating additional background knowledge which results from, e.g., configuration knowledge. Moreover, to keep the inconsistency management problem as flexible as possible, semantic abstraction mechanisms are provided by means of mediations to vocabularies that capture common semantic concepts for, e.g., the domain of mechatronic systems. Hence, instead of envisioning a point-to-point mapping or synchronization of models, which requires to implement bi- or uni-directional model transformations, or an integrated system model or exchange formats in terms of a “world model” of automated production systems, which requires the different discipline-specific models to be integrated into a common modelling language or exchange format, the presented framework allows for coupling the models flexibly at the points where inconsistencies are likely to occur.

An evaluation (Chapter 6) of the proposed framework is presented at two distinct application examples: For one, a lab-scale application example is used to show the feasibility of the presented approach (Section 6.3). Based on this application example, the principle applicability for disparate, but linked models of automated production systems is shown. Moreover, an industry-style application example exemplifies, how additional background knowledge is incorporated in the framework (Section 6.4). Nevertheless, the latter also indicated the framework's limitations – whereas structure models are easily manageable with the presented framework, it lacks in support of behavioural models and inconsistencies. For such dynamic analyses, which require, e.g., the execution of models or a formal verification of the system's dynamics, verification or simulation techniques such as the ones proposed in [Bec^{+}15] are envisioned to be used. A comparison of the proposed inconsistency management approach with another one from the related literature reveals that, while facing implementation challenges as tool support is not as sophisticated as within the “traditional” MBSE domain, the concept proposed in this dissertation implies several benefits. First, the incorporation of background knowledge allows for drawing additional conclusions on potential inconsistencies, which are not yet considered in traditional MBSE approaches. Second, the semantic mediation from (more specific) discipline vocabularies to (more abstract) domain or common vocabularies decreases the specification effort, as the amount of necessary inconsistency rules can be lowered. Finally, an assessment of the performance and scalability of the proposed framework reveals that, with the computing capabilities of current Personal Computers (PCs), results can be achieved within appropriate computing times.

Although, with the proposed framework, a huge step towards managing inconsistencies in heterogeneous models of automated production systems is made, the discussion of its strengths and limitations (Chapter 7) reveals that a lot of research effort is still required to apply such a framework to industrial application. One aspect that needs to be addressed is the additional expert knowledge that is necessary, e.g., in the field of Semantic Web Technologies, which hampers applications in industrial practice. To overcome this drawback, appropriate techniques to specify the multitude of inconsistency diagnosis and handling rules more easily, e.g., by means of graphical patterns (see [Gue^{+}13; Fel^{+}16c]), as well as for managing the multitude of different types of inconsistencies, e.g., by means of supporting model dependency visualization techniques (see [Bas^{+}15]), are required. Such visualization approaches are envisioned to be complimentary to an inconsistency management framework, thereby aiding stakeholders in specifying inconsistency (diagnosis and/or handling) rules, as well as identifying and understanding the inconsistent parts of the engineering solution. Another aspect that is not in focus of this dissertation is the need to incorporate engineering documents. Whereas models certainly simplify the engineering of automated production systems, they cannot entirely replace the documents that are used, e.g., by technicians during commission or maintenance. Consequently, appropriate mechanisms and frameworks must be employed that allow for ensuring the consistency between these documents and the respective engineering models. Finally, the essential restriction of the proposed framework is its focus on models that are compliant to the Meta Object Facility (MOF) [OMG15a]. Accordingly, additional model types that stem from non-technical disciplines such as socio-technological or psychological sciences, cannot be considered in the framework's current form. These aspects, which are essential factors to be considered for innovation processes in the automated production systems domain, must be investigated and included in inconsistency management frameworks in future research efforts.

Bibliography

[3SS16] **3S-Smart Software Solutions GmbH**. *CODESYS Application Composer*. Online. 2016. URL: `https://www.codesys.com/products/codesys-engineering/application-composer.html` (visited on 10/06/2016).

[Abe+13] **Abele, L., Legat, C., Grimm, S., and Müller, A. W.** "Ontology-based validation of plant models". In: *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. 2013, pp. 236–241. DOI: `10.1109/INDIN.2013.6622888`.

[Ahm+17] **Ahmad, M., Ahmad, B., Harrison, R., Ferrer, B. R., and Lastra, J. L. M.** "Ensuring the consistency between assembly process planning and machine control software". In: *IEEE International Conference on Industrial Informatics*. Emden, Germany, 2017. DOI: `10.1109/INDIN.2017.8104923`.

[AS10] **Akerkar, R. and Sajja, P.** *Knowledge-based Systems*. Burlington, MA, USA: Jones & Bartlett Learning, 2010.

[ABS18] **Ananieva, S., Burger, E., and Stier, C.** "Model-Driven Consistency Preservation in AutomationML". In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018. DOI: `10.1109/coase.2018.8560343`.

[Ana+18] **Ananieva, S., Klare, H., Burger, E., and Reussner, R.** "Variants and Versions Management for Models with Integrated Consistency Preservation". In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems - VAMOS 2018*. ACM Press, 2018. DOI: `10.1145/3168365.3168377`.

[Apa16] **Apache Jena**. *Fuseki: serving RDF data over HTTP*. Online. 2016. URL: `https://jena.apache.org/documentation/serving_data/` (visited on 11/11/2016).

[ARC15] **ARC Advisory Group**. *Programmable Logic Controllers (PLCs) and PLC-based Programmable Automation Controllers (PACs)*. Online. 2015. URL: `http://www.arcweb.com/market-studies/pages/plcs-programmable-logic-controllers.aspx` (visited on 04/10/2016).

[Arr+16] **Arroyo, E., Hoernicke, M., Rodríguez, P., and Fay, A.** "Automatic derivation of qualitative plant simulation models from legacy piping and instrumentation diagrams". In: *Computers & Chemical Engineering*, vol. 92 (2016), pp. 112–132. DOI: `10.1016/j.compchemeng.2016.04.040`.

[BV06] **Balogh, A. and Varró, D.** "Advanced Model Transformation Language Constructs in the VIATRA2 Framework". In: *ACM Symposium on Applied Computing*. Dijon, France, 2006, pp. 1280–1287. DOI: `10.1145/1141277.1141575`.

[BFB14] **Barbieri, G., Fantuzzi, C., and Borsari, R.** "A Model-based Design Methodology for the Development of Mechatronic Systems". In: *Mechatronics*, vol. 24, no. 7 (2014), pp. 833–843. DOI: `10.1016/j.mechatronics.2013.12.004`.

[Bas+15] **Basole, R. C., Qamar, A., Park, H., Paredis, C. J. J., and McGinnis, L. F.** "Visual Analytics for Early-Phase Complex Engineered System Design Support". In: *IEEE Computer Graphics and Applications*, vol. 35, no. 2 (2015), pp. 41–51. DOI: `10.1109/MCG.2015.3`.

[Bas+11] **Bassi, L., Secchi, C., Bonfe, M., and Fantuzzi, C.** "A SysML-Based Methodology for Manufacturing Machinery Modeling and Design". In: *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 6 (2011), pp. 1049–1062. DOI: 10.1109/TMECH.2010.2073480.

[Bec+15] **Beckert, B., Ulbrich, M., Vogel-Heuser, B., and Weigl, A.** "Regression Verification for Programmable Logic Controller Software". In: *International Conference on Formal Engineering Methods.* 2015, pp. 234–251. DOI: 10.1007/978-3-319-25423-4_15.

[Ber+16] **Berardinelli, L., Biffl, S., Lüder, A., Mätzler, E., Mayerhofer, T., Wimmer, M., and Wolny, S.** "Cross-disciplinary engineering with AutomationML and SysML". In: *Automatisierungstechnik (at)*, vol. 64, no. 4 (2016), pp. 253–269. DOI: 10.1515/auto-2015-0076.

[Bif+14a] **Biffl, S., Kovalenko, O., Lüder, A., Schmidt, N., and Rosendahl, R.** "Semantic mapping support in AutomationML". In: *IEEE International Conference on Emerging Technology and Factory Automation.* Barcelona, Spain, 2014, pp. 1–4. DOI: 10.1109/ETFA.2014.7005276.

[Bif+15] **Biffl, S., Maetzler, E., Wimmer, M., Lüder, A., and Schmidt, N.** "Linking and versioning support for AutomationML: A model-driven engineering perspective". In: *IEEE International Conference on Industrial Informatics.* Cambridge, UK, 2015, pp. 499–506. DOI: 10.1109/INDIN.2015.7281784.

[BSZ09] **Biffl, S., Schatten, A., and Zoitl, A.** "Integration of heterogeneous engineering environments for the automation systems lifecycle". In: *2009 7th IEEE International Conference on Industrial Informatics.* 2009, pp. 576–581. DOI: 10.1109/INDIN.2009.5195867.

[Bif+14b] **Biffl, S., Winkler, D., Mordinyi, R., Scheiber, S., and Holl, G.** "Efficient monitoring of multi-disciplinary engineering constraints with semantic data integration in the Multi-Model Dashboard process". In: *IEEE International Conference on Emerging Technology and Factory Automation.* 2014, pp. 1–10. DOI: 10.1109/ETFA.2014.7005211.

[BFS13] **Bonfé, M., Fantuzzi, C., and Secchi, C.** "Design Patterns for Model-based Automation Software Design and Implementation". In: *Control Engineering Practice*, vol. 21, no. 11 (2013), pp. 1608–1619. DOI: 10.1016/j.conengprac.2012.03.017.

[Bro+10] **Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., and Ratiu, D.** "Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments". In: *Proceedings of the IEEE*, vol. 98, no. 4 (2010), pp. 526–545. DOI: 10.1109/JPROC.2009.2037771.

[BGT05] **Burmester, S., Giese, H., and Tichy, M.** "Model-Driven Development of Reconfigurable Mechatronic Systems with MECHATRONIC UML". In: *Model-Driven Architecture.* Ed. by **Aßmann, U., Aksit, M., and Rensink, A.** Vol. 3599. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2005, pp. 47–61. DOI: 10.1007/11538097_4.

[CLP11] **Cao, Y., Liu, Y., and Paredis, C. J.** "System-level model integration of design and simulation for mechatronic systems based on SysML". In: *Mechatronics*, vol. 21, no. 6 (2011), pp. 1063–1075. DOI: 10.1016/j.mechatronics.2011.05.003.

[Car$^+$16] **Carlsson, O., Vera, D., Delsing, J., Ahmad, B., and Harrison, R.** "Plant descriptions for engineering tool interoperability". In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. 2016, pp. 730–735. DOI: `10.1109/INDIN.2016.7819255`.

[Dem$^+$16] **Demuth, A., Kretschmer, R., Egyed, A., and Maes, D.** "Introducing Traceability and Consistency Checking for Change Impact Analysis across Engineering Tools in an Automation Solution Company: An Experience Report". In: *IEEE International Conference on Software Maintenance and Evolution*. 2016. DOI: `10.1109/icsme.2016.50`.

[DM13] **Dickerson, C. E. and Mavris, D.** "A Brief History of Models and Model Based Systems Engineering and the Case for Relational Orientation". In: *IEEE Systems Journal*, vol. 7, no. 4 (2013), pp. 581–592. DOI: `10.1109/JSYST.2013.2253034`.

[Dra10] **Drath, R.**, ed. *Datenaustausch in der Anlagenplanung mit AutomationML*. Berlin, Heidelberg, Germany: Springer, 2010. DOI: `10.1007/978-3-642-04674-2`.

[eCl16] **eCl@ss e.V.** *The eCl@ss standard: Classification and product description*. Online. 2016. URL: `http://www.eclass.eu/eclasscontent/standard/index.html.en` (visited on 10/31/2016).

[Ecl15] **Eclipse Foundation**. *Papyrus Modeling Environment*. Online. 2015. URL: `https://eclipse.org/papyrus/` (visited on 04/10/2016).

[Ecl16a] **Eclipse Foundation**. *Acceleo Model To Text (M2T)*. Online. 2016. URL: `http://www.eclipse.org/acceleo/` (visited on 11/11/2016).

[Ecl16b] **Eclipse Foundation**. *Eclipse Epsilon Framework*. Online. 2016. URL: `http://www.eclipse.org/epsilon/` (visited on 10/06/2016).

[Ecl16c] **Eclipse Foundation**. *Eclipse Modeling Framework*. Online. 2016. URL: `http://www.eclipse.org/modeling/emf/` (visited on 10/06/2016).

[Egy11] **Egyed, A.** "Automatically Detecting and Tracking Inconsistencies in Software Design Models". In: *IEEE Transactions on Software Engineering*, vol. 37, no. 2 (2011), pp. 188–204. DOI: `10.1109/TSE.2010.38`.

[Egy$^+$18] **Egyed, A., Zeman, K., Hehenberger, P., and Demuth, A.** "Maintaining Consistency across Engineering Artifacts". In: *Computer*, vol. 51, no. 2 (2018), pp. 28–35. DOI: `10.1109/mc.2018.1451666`.

[Eka$^+$17] **Ekaputra, F. J., Sabou, M., Serral, E., Kiesling, E., and Biffl, S.** "Ontology-Based Data Integration in Multi-Disciplinary Engineering Environments: A Review". In: *Open Journal of Information Systems*, vol. 4, no. 1 (2017), pp. 1–26.

[ElM06] **ElMaraghy, H. A.** "Flexible and Reconfigurable Manufacturing Systems Paradigms". In: *International Journal of Flexible Manufacturing Systems*, vol. 17, no. 4 (2006), pp. 261–276. DOI: `10.1007/s10696-006-9028-7`.

[EPL16] **EPLAN Software & Service GmbH & Co. KG**. *EPLAN Engineering Configuration*. Online. 2016. URL: `http://www.engineeringconfiguration.com/en/eec-home` (visited on 12/02/2016).

[EM12] **Estévez, E. and Marcos, M.** "Model-Based Validation of Industrial Control Systems". In: *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2 (2012), pp. 302–310. DOI: `10.1109/TII.2011.2174248`.

[EMO07] **Estêvez, E., Marcos, M., and Orive, D.** "Automatic Generation of PLC Automation Projects from Component-based Models". In: *The International Journal of Advanced Manufacturing Technology*, vol. 35, no. 5 (2007), pp. 527–540. DOI: `10.1007/s00170-007-1127-4`.

[Fay+17] **Fay, A., Scholz, A., Hildebrandt, C., Schröder, T., Diedrich, C., Dubovy, M., Wiegand, R., Eck, C., and Heidel, R.** *Semantische Inhalte für Industrie 4.0: Modellierung technischer Systeme in kollaborativen Umgebungen*. German. In: *atp edition – Automatisierungstechnische Praxis*, vol. 59, 7–9 (2017), pp. 34–43.

[Fel+16a] **Feldmann, S., Hauer, F., Ulewicz, S., and Vogel-Heuser, B.** "Analysis Framework for Evaluating PLC Software: An Application of Semantic Web Technologies". In: *IEEE International Symposium on Industrial Electronics*. Santa Clara, CA, USA, 2016.

[Fel+19] **Feldmann, S., Kernschmidt, K., Wimmer, M., and Vogel-Heuser, B.** "Managing Inter-Model Inconsistencies in Model-based Systems Engineering: Application in Automated Production Systems Engineering". In: *Journal of Systems and Software*, vol. 153 (2019), pp. 105–134. DOI: `10.1016/j.jss.2019.03.060`.

[Fel+16b] **Feldmann, S., Ulewicz, S., Diehm, S., and Vogel-Heuser, B.** *Strukturelle Codeanalyse: Analyseframework mittels Semantic-Web-Technologien*. German. In: *atp edition*, vol. 58, 9 (2016), pp. 42–51.

[Fel+16c] **Feldmann, S., Wimmer, M., Kernschmidt, K., and Vogel-Heuser, B.** "A Comprehensive Approach for Managing Inter-Model Inconsistencies in Automated Production Systems Engineering". In: *IEEE International Conference on Automation Science and Engineering*. Fort Worth, TX, USA, 2016.

[FFV12] **Feldmann, S., Fuchs, J., and Vogel-Heuser, B.** "Modularity, Variant and Version Management in Plant Automation – Future Challenges and State of the Art". In: *International Design Conference*. Dubrovnik, Croatia, 2012, pp. 1689–1698. URL: `https://www.designsociety.org/publication/32138/modularity_variant_and_version_management_in_plant_automation_%E2%80%93_future_challenges_and_state_of_the_art` (visited on 05/20/2019).

[Fel+17] **Feldmann, S., Hauer, F., Pantförder, D., Pankratz, F., Klinker, G., and Vogel-Heuser, B.** "Management of Inconsistencies in Domain-Spanning Models – An Interactive Visualization Approach". In: *19th International Conference on Human-Computer Interaction*. Vancouver, Canada, 2017.

[Fel+15a] **Feldmann, S., Herzig, S. J. I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C. J. J., and Vogel-Heuser, B.** "A Comparison of Inconsistency Management Approaches Using a Mechatronic Manufacturing System Design Case Study". In: *IEEE International Conference on Automation Science and Engineering*. Gothenburg, Sweden, 2015, pp. 158–165. DOI: `10.1109/CoASE.2015.7294055`.

[Fel+15b] **Feldmann, S., Herzig, S. J. I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C. J. J., and Vogel-Heuser, B.** "Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems". In: *15th IFAC Symposium on Information Control Problems in Manufacturing*. Ottawa, Canada, 2015, pp. 916–923. DOI: `10.1016/j.ifacol.2015.06.200`.

[FKV16] **Feldmann, S., Kernschmidt, K., and Vogel-Heuser, B.** "Applications of Semantic Web Technologies for the Engineering of Automated Production Systems – Three Use Cases". In: *Semantic Web Technologies in Intelligent Engineering Applications*. Ed. by **Biffl, S. and Sabou, M.** Berlin, Heidelberg, Germany: Springer, 2016.

[Fin+94] **Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B.** "Inconsistency Handling in Multiperspective Specifications". In: *IEEE Transactions on Software Engineering*, vol. 20, no. 8 (1994), pp. 569–578. DOI: `10.1109/32.310667`.

[Fis16] **Fischer, J.** *Development, Application and Evaluation of an Inconsistency Management Approach for Interdisciplinary Engineering with a Special Focus on Electrical Engineering.* Technical University of Munich, 2016.

[for16] **fortiss GmbH**. *AF3 – Seamless Model-based Development – From requirements analysis to platform.* Online. 2016. URL: `http://af3.fortiss.org/` (visited on 10/25/2016).

[Gau+07] **Gausemeier, J., Giese, H., Schäfer, W., Axenath, B., Frank, U., Henkler, S., Pook, S., and Tichy, M.** "Towards the design of self-optimizing mechatronic systems: Consistency between domain-spanning and domain-specific models". In: *ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference.* 2007, pp. 1141–1148.

[Gau+09] **Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., and Rieke, J.** "Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems". In: *International Conference on Engineering Design.* Palo Alto, CA, USA, 2009, pp. 1–12.

[GS13] **Giese, H. and Schäfer, W.** "Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML". In: *Assurances for Self-Adaptive Systems.* Ed. by **Cámara, J., Lemos, R., Ghezzi, C., and Lopes, A.** Vol. 7740. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2013, pp. 152–186. DOI: `10.1007/978-3-642-36249-1_6`.

[GW06] **Giese, H. and Wagner, R.** "Incremental Model Synchronization with Triple Graph Grammars". In: *International Conference on Model Driven Engineering Languages and Systems.* Genova, Italy, 2006, pp. 543–557. DOI: `10.1007/11880240_38`.

[GW09] **Giese, H. and Wagner, R.** "From model transformation to incremental bidirectional model synchronization". In: *Software & Systems Modeling*, vol. 8, no. 1 (2009), pp. 21–43. DOI: `10.1007/s10270-008-0089-9`.

[GF16] **Glawe, M. and Fay, A.** "Wissensbasiertes Engineering automatisierter Anlagen unter Verwendung von AutomationML und OWL". In: *Automatisierungstechnik (at)*, vol. 64, no. 3 (2016), pp. 186–198. DOI: `10.1515/auto-2015-0077`.

[Gla+15] **Glawe, M., Tebbe, C., Fay, A., and Niemann, K.-H.** "Knowledge-based Engineering of Automation Systems using Ontologies and Engineering Data". In: *International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.* 2015, pp. 291–300. DOI: `10.5220/0005614502910300`.

[Gue+13] **Guerra, E., Lara, J. de, Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W.** "Automated Verification of Model Transformations based on Visual Contracts". In: *Autom. Softw. Eng.* Vol. 20, no. 1 (2013), pp. 5–46.

[HD13] **Hadlich, T. and Diedrich, C.** "Using properties in systems engineering". In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA).* 2013. DOI: `10.1109/ETFA.2013.6647979`.

[Hau16] **Hauer, F.** *Development of a Framework and an Interactive Visualization Approach for Supporting the Identification and Resolution of Inconsistencies in Domain-spanning Models.* Technical University of Munich, 2016.

[Heg+11] **Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M. C., and Varró, D.** "Quick fix generation for DSMLs". In: *IEEE Symposium on Visual Languages and Human-Centric Computing.* Pittsburgh, PA, USA, 2011, pp. 17–24. DOI: `10.1109/VLHCC.2011.6070373`.

[HEZ10] **Hehenberger, P., Egyed, A., and Zeman, K.** "Consistency Checking of Mechatronic Design Models". In: *ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference.* Montreal, Quebec, Canada, 2010, pp. 1141–1148. DOI: `10.1115/DETC2010-28615`.

[Her15] **Herzig, S.** "A Bayesian Learning Approach to Inconsistency Identification in Model-based Systems Engineering". Ph.D. Dissertation. Atlanta, Georgia, USA: Georgia Institute of Technology, 2015. URL: `https://smartech.gatech.edu/bitstream/handle/1853/53576/HERZIG-DISSERTATION-2015.pdf` (visited on 04/10/2016).

[Her+11] **Herzig, S., Qamar, A., Reichwein, A., and Paredis, C.** "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, 2011, pp. 1329–1339. DOI: `10.1115/DETC2011-47924`.

[HP14] **Herzig, S. J. and Paredis, C. J.** "A Conceptual Basis for Inconsistency Management in Model-based Systems Engineering". In: *CIRP Design Conference.* Milano, Italy, 2014, pp. 52–57. DOI: `10.1016/j.procir.2014.03.192`.

[HQP14] **Herzig, S. J., Qamar, A., and Paredis, C. J.** "An Approach to Identifying Inconsistencies in Model-based Systems Engineering". In: *Conference on Systems Engineering Research.* Redondo Beach, CA, USA, 2014, pp. 354–362. DOI: `10.1016/j.procs.2014.03.044`.

[Hir+02] **Hirtz, J., Stone, R. B., McAdams, D. A., Szykman, S., and Wood, K. L.** *A Functional Basis for Engineering Design: Reconciling and Evolving Previous Efforts.* Tech. rep. NIST Technical Note 1447. National Institute of Standards and Technology, 2002. URL: `https://www.nist.gov/node/742436` (visited on 12/02/2016).

[HKR10] **Hitzler, P., Krötzsch, M., and Rudolph, S.** *Foundations of Semantic Web Technologies.* Boca Raton, FL, USA: CRC Press, 2010.

[Huz+05] **Huzar, Z., Kuzniarz, L., Reggio, G., and Sourrouille, J. L.** "Consistency Problems in UML-Based Software Development". In: *UML Modeling Languages and Applications.* Ed. by **Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., and Toval Alvarez, A.** Vol. 3297. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2005, pp. 1–12. DOI: `10.1007/978-3-540-31797-5_1`.

[IEC03a] **IEC**. *Programmable Controllers – Part 1: General Information.* IEC Standard IEC 61131-1:2003. 2003.

[IEC03b] **IEC**. *Programmable Controllers – Part 3: Programming Languages.* IEC Standard IEC 61131-3:2003. 2003.

[IEC12] **IEC**. *Function Blocks – Part 1: Architecture.* IEC Standard IEC 61499-1:2012. 2012.

[IEC13a] **IEC**. *International Electrotechnical Vocabulary – Part 351: Control Technology.* IEC Standard IEC 60050-351. 2013.

[IEC13b] **IEC**. *Programmable Controllers – Part 3: Programming Languages.* IEC Standard IEC 61131-3:2013. 2013.

[IEC14] **IEC**. *Engineering Data Exchange Format for Use in Industrial Automation Systems Engineering – Automation Markup Language – Part 1: Architecture and General Requirements.* IEC Standard IEC 62714-1. 2014.

[IEC16] **IEC**. *Representation of Process Control Engineering – Requests in P&I Diagrams and Data Exchange Between P&ID Tools and PCE-CAE Tools.* IEC Standard IEC 62424:2016. 2016.

[INC07] **INCOSE**. *Systems Engineering Vision 2020.* Tech. rep. INCOSE-TP-2004-004-02. 2007. URL: `http://oldsite.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf` (visited on 12/02/2016).

[ISA10] **ISA**. *Enterprise-Control System Integration – Part 1: Models and Terminology.* ISA Standard ANSI/ISA-95.00.01-2010. 2010.

[Jäg+12] **Jäger, T., Fay, A., Wagner, T., and Löwen, U.** "Comparison of engineering results within domain specific languages regarding information contents and intersections". In: *International Multi-Conference on Systems, Signals and Devices.* 2012, pp. 1–6. DOI: `10.1109/SSD.2012.6197913`.

[Jou+08] **Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I.** "ATL: A model transformation tool". In: *Science of Computer Programming*, vol. 72, no. 1-2 (2008), pp. 31–39. DOI: `10.1016/j.scico.2007.08.002`.

[Kam+15] **Kammerl, D., Malaschewski, O., Schenkl, S. A., and Mörtl, M.** "Decision Uncertainties in the Planning of Product-Service System Portfolios". In: *5th International Conference on Research into Design.* 2015, pp. 39–48.

[Kat+19] **Kattner, N., Bauer, H., Basirati, M., Zou, M., Vogel-Heuser, B., Böhm, M., Krcmar, H., Reinhart, G., and Lindemann, U.** "Inconsistency management in heterogeneous models – an approach for the identification of model dependencies and potential inconsistencies". In: *International Conference on Engineering Design.* 2019.

[Ker17] **Kernschmidt, K.** "Interdisciplinary Structural Modeling of Mechatronic Production Systems Using SysML4*Mechatronics*". Dr.-Ing. Dissertation. Munich, Germany: Technical University of Munich, 2017.

[KV13] **Kernschmidt, K. and Vogel-Heuser, B.** "An Interdisciplinary SysML Based Modeling Approach for Analyzing Change Influences in Production Plants to Support the Engineering". In: *IEEE International Conference on Automation Science and Engineering.* Madison, WI, USA, 2013, pp. 1113–1118. DOI: `10.1109/CoASE.2013.6654030`.

[Ker+13] **Kernschmidt, K., Wolfenstetter, T., Münzberg, C., Kammerl, D., Goswami, S., Lindemann, U., Krcmar, H., and Vogel-Heuser, B.** "Concept for an integration-framework to enable the crossdisciplinary development of product-service systems". In: *2013 IEEE International Conference on Industrial Engineering and Engineering Management.* 2013, pp. 340–345. DOI: `10.1109/IEEM.2013.6962430`.

[Ker+14] **Kernschmidt, K., Behncke, F., Chucholowski, N., Wickel, M., Bayrak, G., Lindemann, U., and Vogel-Heuser, B.** "An Integrated Approach to Analyze Change-situations in the Development of Production Systems". In: *CIRP Conference on Manufacturing Systems.* Vol. 17. 2014, pp. 148–153. DOI: `10.1016/j.procir.2014.01.081`.

[KP09] **Kerzhner, A. A. and Paredis, C. J. J.** "Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering". In: *ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference.* 2009. DOI: `10.1115/DETC2009-87286`.

[KP10] **Kerzhner, A. A. and Paredis, C. J. J.** "Model-Based System Verification: A Formal Framework for Relating Analyses, Requirements, and Tests". In: *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2010. DOI: 10.1007/978-3-642-21210-9_27.

[Kol+17] **Koltun, G. D., Feldmann, S., Schütz, D., and Vogel-Heuser, B.** "Model-Document Coupling in aPS Engineering: Challenges and Requirements Engineering Use Case". In: *18th IEEE International Conference on Industrial Technology*. 2017. DOI: 10.1109/ICIT.2017.7915529.

[Kon18] **Konersmann, M.** "Explicitly Integrated Architecture: An Approach for Integrating Software Architecture Model Information with Program Code". PhD thesis. Universität Duisburg-Essen, 2018.

[KG12] **Konersmann, M. and Goedicke, M.** "A Conceptual Framework and Experimental Workbench for Architectures". In: *Software Service and Application Engineering*. Springer Berlin Heidelberg, 2012, pp. 36–52. DOI: 10.1007/978-3-642-30835-2_4.

[Kov+15] **Kovalenko, O., Wimmer, M., Sabou, M., Lüder, A., Ekaputra, F. J., and Biffl, S.** "Modeling AutomationML: Semantic Web technologies vs. Model-Driven Engineering". In: *IEEE Conference on Emerging Technologies Factory Automation*. Luxembourg, 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301643.

[Kov+14a] **Kovalenko, O., Serral, E., Sabou, M., Ekaputra, F. J., Winkler, D., and Biffl, S.** "Automating Cross-Disciplinary Defect Detection in Multi-disciplinary Engineering Environments". In: *International Conference on Knowledge Engineering and Knowledge Management*. Linköping, Sweden, 2014, pp. 238–249. DOI: 10.1007/978-3-319-13704-9_19.

[Kov+14b] **Kovalenko, O., Winkler, D., Kalinowski, M., Serral, E., and Biffl, S.** "Engineering Process Improvement in Heterogeneous Multi-disciplinary Environments with Defect Causal Analysis". In: *European Conference on Systems, Software and Services Process Improvement*. 2014, pp. 73–85. DOI: 10.1007/978-3-662-43896-1_7.

[KBL13] **Kramer, M. E., Burger, E., and Langhammer, M.** "View-centric engineering with synchronized heterogeneous models". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13*. ACM Press, 2013. DOI: 10.1145/2489861.2489864.

[Kra+15] **Kramer, M. E., Langhammer, M., Messinger, D., Seifermann, S., and Burger, E.** "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM Press, 2015. DOI: 10.1145/2737166.2737177.

[Kra17] **Kramer, M. E.** "Specification Languages for Preserving Consistency between Models of Different Languages". en. In: (2017). DOI: 10.5445/ir/1000069284.

[LG99] **Lauber, R. and Göhner, P.** *Prozessautomatisierung 1*. 3rd ed. Berlin, Heidelberg, Germany: Springer, 1999. DOI: 10.1007/978-3-642-58446-6.

[Leg+14] **Legat, C., Mund, J., Campetelli, A., Hackenberg, G., Folmer, J., Schütz, D., Broy, M., and Vogel-Heuser, B.** "Interface Behavior Modeling for Automatic Verification of Industrial Automation Systems' Functional Conformance". In: *at – Automatisierungstechnik*, vol. 62, no. 11 (2014), pp. 815–825. DOI: 10.1515/auto-2014-1126.

[Lei13] **Leitão, P.** "Multi-agent Systems in Industry: Current Trends & Future Challenges". In: *Beyond Artificial Intelligence*. Ed. by **Kelemen, J., Romportl, J., and Zackova, E.** Vol. 4. Topics in Intelligent Engineering and Informatics. Berlin, Heidelberg, Germany: Springer, 2013, pp. 197–201. DOI: 10.1007/978-3-642-34422-0_13.

[LCK15] **Leitão, P., Colombo, A. W., and Karnouskos, S.** "Industrial Automation Based on Cyber-physical Systems Technologies: Prototype Implementations and Challenges". In: *Computers in Industry* (2015). DOI: 10.1016/j.compind.2015.08.004.

[Lin+15a] **Lin, H. Y., Sierla, S., Papakonstantinou, N., Shalyto, A., and Vyatkin, V.** "Change request management in model-driven engineering of industrial automation software". In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. 2015, pp. 1186–1191. DOI: 10.1109/INDIN.2015.7281904.

[Lin+15b] **Lin, H. Y., Sierla, S., Papakonstantinou, N., and Vyatkin, V.** "A SysML profile supporting change orders in model driven engineering". In: *2015 IEEE International Conference on Automation Science and Engineering (CASE)*. 2015, pp. 1054–1059. DOI: 10.1109/CoASE.2015.7294238.

[LPW18] **Lüder, A., Pauly, J., and Wimmer, M.** "Modelling consistency rules within production system engeering". In: *14th International Conference on Automation Science and Engineering*. 2018, pp. 664–667. DOI: 10.1109/COASE.2018.8560537.

[MJ12] **Maga, C. and Jazdi, N.** "Interdisciplinary Modularization in Product Line Engineering: A Case Study". In: *IEEE International Conference on Automation Quality and Testing Robotics*. Cluj-Napoca, Romania, 2012, pp. 179–184. DOI: 10.1109/AQTR.2012.6237699.

[Mar+09] **Marcos, M., Estévez, E., Perez, F., and Wal, E. V. D.** "XML Exchange of Control Programs". In: *IEEE Industrial Electronics Magazine*, vol. 3, no. 4 (2009), pp. 32–35. DOI: 10.1109/MIE.2009.934794.

[MSD06] **Mens, T., Straeten, R., and D'Hondt, M.** "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis". In: *International Conference on Model Driven Engineering Languages and Systems*. Genova, Italy, 2006, pp. 200–214. DOI: 10.1007/11880240_15.

[MVS05] **Mens, T., Van Der Straeten, R., and Simmonds, J.** "A Framework for Managing Consistency of Evolving UML Models". In: *Software Evolution with UML and XML*. Idea Group Publishing, 2005.

[Mor+12] **Mordinyi, R., Moser, T., Winkler, D., and Biffl, S.** "Navigating between tools in heterogeneous Automation Systems Engineering landscapes". In: *Annual Conference on IEEE Industrial Electronics Society*. 2012, pp. 6178–6184. DOI: 10.1109/IECON.2012.6389070.

[MSB15] **Mordinyi, R., Schindler, P., and Biffl, S.** "Evaluation of NoSQL graph databases for querying and versioning of engineering data in multi-disciplinary engineering environments". In: *IEEE International Conference on Emerging Technologies Factory Automation*. 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301486.

[Mor+14] **Mordinyi, R., Serral, E., Winkler, D., and Biffl, S.** "Evaluating software architectures using ontologies for storing and versioning of engineering data in heterogeneous systems engineering environments". In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 2014, pp. 1–10. DOI: 10.1109/ETFA.2014.7005237.

[MB12] **Moser, T. and Biffl, S.** "Semantic Integration of Software and Systems Engineering Environments". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1 (2012), pp. 38–50. DOI: `10.1109/TSMCC.2011.2136377`.

[Mos⁺11] **Moser, T., Biffl, S., Sunindyo, W., and Winkler, D.** "Integrating Production Automation Expert Knowledge Across Engineering Domains". In: *International Journal of Distributed Systems and Technologies*, vol. 2, no. 3 (2011), pp. 88–103. DOI: `10.4018/jdst.2011070106`.

[MSD10] **Muehlhause, M., Suchold, N., and Diedrich, C.** "Application of semantic technologies in engineering processes for manufacturing systems". In: *IFAC Workshop on Intelligent Manufacturing Systems*. Lisbon, Portugal, 2010. DOI: `10.3182/20100701-2-PT-4011.00011`.

[NAM10] **NAMUR**. *Use of Lists of Properties in Process Control Engineering Workflows*. NAMUR Recommendation NE 100 Version 3.2. 2010.

[NAS00] **NASA**. *Report on Project Management in NASA: Phase II of the Mars Climate Orbiter Mishap Report*. 2000. URL: `ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/MCO_MIB_Report.pdf` (visited on 10/31/2016).

[NAS16] **NASA**. *Quantities, Units, Dimensions and Types Catalog Release 1.1*. Online. 2016. URL: `http://www.linkedmodel.org/catalog/qudt/1.1/` (visited on 10/31/2016).

[Neu94] **Neufville, R. de**. "The baggage system at Denver: prospects and lessons". In: *Journal of Air Transport Management*, vol. 1, no. 4 (1994), pp. 229–236. DOI: `10.1016/0969-6997(94)90014-0`.

[NoM16] **NoMagic, Inc.** *MagicDraw*. Online. 2016. URL: `http://www.nomagic.com/products/magicdraw.html` (visited on 04/10/2016).

[NER00] **Nuseibeh, B., Easterbrook, S., and Russo, A.** "Leveraging Inconsistency in Software Development". In: *IEEE Computer*, vol. 33, no. 4 (2000), pp. 24–29. DOI: `10.1109/2.839317`.

[Obj08] **Object Management Group**. *MOF Model To Text Transformation Language Version 1.0*. Online. 2008. URL: `http://www.omg.org/spec/MOFM2T/1.0/` (visited on 11/03/2016).

[Obj14] **Object Management Group**. *Model Driven Architecture (MDA) – MDA Guide 2.0*. Online. 2014. URL: `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01` (visited on 11/03/2016).

[Obj15] **Object Management Group**. *XML Metadata Interchange (XMI) Specification Version 2.5.1*. Online. 2015. URL: `http://www.omg.org/spec/XMI/2.5.1` (visited on 11/03/2016).

[Obj16] **Object Management Group**. *MOF 2.0 Query/View/Transformation Specification Version 1.3*. Online. 2016. URL: `http://www.omg.org/spec/QVT/1.3/` (visited on 11/03/2016).

[OMG11] **OMG**. *UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems Version 1.1*. Online. 2011. URL: `http://www.omg.org/spec/MARTE/1.1/` (visited on 04/10/2016).

[OMG15a] **OMG**. *Meta Object Facility (MOF) Version 2.5*. Online. 2015. URL: `www.omg.org/spec/MOF/2.5/` (visited on 04/10/2016).

[OMG15b] **OMG**. *Systems Modeling Language Version 1.4*. Online. 2015. URL: `http://www.omg.org/spec/SysML/1.4/` (visited on 04/10/2016).

[OMG15c] **OMG**. *Unified Modeling Language Version 2.5*. Online. 2015. URL: `http://www.omg.org/spec/UML/2.5/` (visited on 04/10/2016).

[PLC16] **PLCopen Promotional Committee 2**. *PLCopen Promotional Committee Training: Coding Guidelines*. 2016. URL: `http://www.plcopen.org/pages/pc2_training/` (visited on 04/10/2016).

[PLC09a] **PLCopen Technical Committee 3**. *PLCopen Guideline: Compliance Testing & Certification*. 2009. URL: `http://www.plcopen.org/pages/tc3_certification/` (visited on 04/10/2016).

[PLC09b] **PLCopen Technical Committee 6**. *XML Formats for IEC 61131-3 V2.01*. 2009. URL: `http://www.plcopen.org/pages/tc6_xml/` (visited on 04/10/2016).

[PTC16] **PTC**. *PTC Integrity Modeler*. Online. 2016. URL: `https://www.ptc.com/model-based-systems-engineering/integrity-modeler` (visited on 11/03/2016).

[pur16] **pure-systems GmbH**. *pure::variants: Variant Management with pure::variants*. Online. 2016. URL: `https://www.pure-systems.com/products/pure-variants-9.html` (visited on 10/10/2016).

[Qam+12] **Qamar, A., Paredis, C., Wikander, J., and During, C.** "Dependency Modeling and Model Management in Mechatronic Design". In: *Journal of Computing and Information Science in Engineering*, vol. 12, no. 4 (2012). DOI: `10.1115/1.4007986`.

[QWD15] **Qamar, A., Wikander, J., and During, C.** "Managing dependencies in mechatronic design: a case study on dependency management between mechanical design and system design". In: *Engineering with Computers*, vol. 31, no. 3 (2015), pp. 631–646. DOI: `10.1007/s00366-014-0366-x`.

[RE12] **Reder, A. and Egyed, A.** "Computing repair trees for resolving inconsistencies in design models". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. ACM Press, 2012. DOI: `10.1145/2351676.2351707`.

[RP11] **Reichwein, A. and Paredis, C.** "Overview of Architecture Frameworks and Modeling Languages for Model-Based Systems Engineering". In: *ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Washington, US-DC, 2011. DOI: `10.1115/DETC2011-48028`.

[Rie+12] **Rieke, J., Dorociak, R., Sudmann, O., Gausemeier, J., and Schäfer, W.** "Management of cross-domain model consistency for behavioral models of mechatronic systems". In: *International Design Conference*. Dubrovnik, Croatia, 2012, pp. 1781–1790.

[RJV09] **Romero, J. R., Jaen, J. I., and Vallecillo, A.** "Realizing Correspondences in Multi-viewpoint Specifications". In: *IEEE International Enterprise Distributed Object Computing Conference*. Auckland, New Zealand, 2009, pp. 163–172. DOI: `10.1109/EDOC.2009.23`.

[Rös16] **Rösch, S.** "Model-based testing of fault scenarios in production automation". PhD thesis. Technical University of Munich, 2016.

[RV17] **Rösch, S. and Vogel-Heuser, B.** "A Light-Weight Fault Injection Approach to Test Automated Production System PLC Software in Industrial Practice". In: *Control Engineering Practice*, vol. 58 (2017), pp. 12–23. DOI: `10.1016/j.conengprac.2016.09.012`.

[RF11] **Runde, S. and Fay, A.** “Software Support for Building Automation Requirements Engineering – An Application of Semantic Web Technologies in Automation”. In: *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4 (2011), pp. 723–730. DOI: 10.1109/TII.2011.2166784.

[Sab+16] **Sabou, M., Ekaputra, F., Kovalenko, O., and Biffl, S.** “Supporting the engineering of cyber-physical production systems with the AutomationML analyzer”. In: *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*. IEEE, 2016. DOI: 10.1109/cpps.2016.7483919.

[SW10] **Schäfer, W. and Wehrheim, H.** “Model-Driven Development with Mechatronic UML”. In: *Graph Transformations and Model-Driven Engineering*. Ed. by **Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., and Westfechtel, B.** Vol. 5765. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2010, pp. 533–554. DOI: 10.1007/978-3-642-17322-6_23.

[Sch+03] **Schätz, B., Braun, P., Huber, F., and Wisspeintner, A.** “Consistency in Model-based Development”. In: *IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. 2003, pp. 287–296. DOI: 10.1109/ECBS.2003.1194810.

[Sch06] **Schmidt, D. C.** “Guest Editor’s Introduction: Model-Driven Engineering”. In: *IEEE Computer*, vol. 39, no. 2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[Sch+14] **Schmidt, N., Lüder, A., Steininger, H., and Biffl, S.** “Analyzing requirements on software tools according to the functional engineering phase in the technical systems engineering process”. In: *IEEE International Conference on Emerging Technology and Factory Automation*. 2014, pp. 1–8. DOI: 10.1109/ETFA.2014.7005144.

[SFJ15] **Schröck, S., Fay, A., and Jäger, T.** “Systematic interdisciplinary reuse within the engineering of automated plants”. In: *IEEE International Systems Conference*. 2015, pp. 508–515. DOI: 10.1109/SYSCON.2015.7116802.

[Sch+15] **Schröck, S., Zimmer, F., Fay, A., and Jäger, T.** “Systematic reuse of interdisciplinary components supported by engineering relations”. In: 2015, pp. 1545–1552. DOI: 10.1016/j.ifacol.2015.06.306.

[Sch+13a] **Schröck, S., Zimmer, F., Holm, T., Fay, A., and Jäger, T.** “Principles, viewpoints and effect links in the engineering of automated plants”. In: *Annual Conference of the IEEE Industrial Electronics Society*. 2013, pp. 6940–6945. DOI: 10.1109/IECON.2013.6700283.

[Sch94] **Schürr, A.** “Specification of graph translators with triple graph grammars”. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Herrsching, Germany, 1994, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.

[Sch+13b] **Schütz, D., Wannagat, A., Legat, C., and Vogel-Heuser, B.** “Development of PLC-Based Software for Increasing the Dependability of Production Automation Systems”. In: *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4 (2013), pp. 2397–2406. DOI: 10.1109/TII.2012.2229285.

[SBF07] **Secchi, C., Bonfe, M., and Fantuzzi, C.** “On the Use of UML for Modeling Mechatronic Systems”. In: *IEEE Transactions on Automation Science and Engineering*, vol. 4, no. 1 (2007), pp. 105–113. DOI: 10.1109/TASE.2006.879686.

[Sel98] **Selic, B.** “Using UML for modeling complex real-time systems”. In: *Languages, Compilers, and Tools for Embedded Systems*. Ed. by **Mueller, F. and Bestavros, A.** Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 250–260. DOI: 10.1007/BFb0057795.

[Ser+13] **Serral, E., Mordinyi, R., Kovalenko, O., Winkler, D., and Biffl, S.** "Evaluation of semantic data storages for integrating heterogenous disciplines in automation systems engineering". In: *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE.* 2013, pp. 6858–6865. DOI: `10.1109/IECON.2013.6700268`.

[Sha+12] **Shah, A., Paredis, C., Burkhart, R., and Schaefer, D.** "Combining Mathematical Programming and SysML for Automated Component Sizing of Hydraulic Systems". In: *Journal of Computing and Information Science in Engineering*, vol. 12, no. 4 (2012). DOI: `10.1115/1.4007764`.

[SSP09] **Shah, A., Schaefer, D., and Paredis, C.** "Enabling Multi-View Modeling With SysML Profiles and Model Transformations". In: *International Conference on Product Lifecycle Management.* 2009.

[Sha+10] **Shah, A. A., Kerzhner, A. A., Schaefer, D., and Paredis, C. J. J.** "Multi-view Modeling to Support Embedded Systems Engineering in SysML". In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday.* Ed. by **Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., and Westfechtel, B.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 580–601. DOI: `10.1007/978-3-642-17322-6_25`.

[Sie16] **Siemens AG**. *COMOS at a glance.* Online. 2016. URL: `http://w3.siemens.com/mcms/plant-engineering-software/en/comos-overview/` (visited on 10/06/2016).

[SZ01] **Spanoudakis, G. and Zisman, A.** "Inconsistency Management in Software Engineering: Survey and Open Research Issues". In: *Handbook of Software Engineering & Knowledge Engineering: Fundamentals.* Ed. by **Chang, S. K.** Vol. 1. Singapore: World Scientific Publishing Co Pte. Ltd., 2001, pp. 329–380.

[Str+09] **Strasser, T., Rooker, M., Hegny, I., Wenger, M., Zoitl, A., Ferrarini, L., Dede, A., and Colla, M.** "A Research Roadmap for Model-driven Design of Embedded Systems for Automation Components". In: *IEEE International Conference on Industrial Informatics.* Cardiff, UK, 2009, pp. 564–569. DOI: `10.1109/INDIN.2009.5195865`.

[Thr05] **Thramboulidis, K.** "Model-integrated Mechatronics – Toward a New Paradigm in the Development of Manufacturing Systems". In: *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1 (2005), pp. 54–61. DOI: `10.1109/TII.2005.844427`.

[Thr10] **Thramboulidis, K.** "The 3+1 SysML View-Model in Model Integrated Mechatronics". In: *Journal of Software Engineering and Applications*, vol. 3, no. 2 (2010), pp. 109–118. DOI: `10.4236/jsea.2010.32014`.

[Thr12] **Thramboulidis, K.** "IEC 61499 as an Enabler of Distributed and Intelligent Automation: A State-of-the-art Review – A Different View". In: *Journal of Engineering*, vol. 2013 (2012). DOI: `10.1155/2013/638521`.

[TF11] **Thramboulidis, K. and Frey, G.** "Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation". In: *Journal of Software Engineering and Applications*, vol. 4, no. 4 (2011), pp. 217–226. DOI: `10.4236/jsea.2011.44024`.

[Thr13] **Thramboulidis, K.** "Overcoming Mechatronic Design Challenges: The 3+1 SysML-view Model". In: *Journal of Computing Science and Technology*, vol. 1, no. 1 (2013), pp. 6–14. URL: `http://researchpub.org/journal/cstij/number/vol1-no1/vol1-no1-1.pdf` (visited on 04/10/2016).

[VD06] **Van Der Straeten, R. and D'Hondt, M.** "Model Refactorings Through Rule-based Inconsistency Resolution". In: *ACM Symposium on Applied Computing.* SAC '06. Dijon, France: ACM, 2006, pp. 1210–1217. DOI: `10.1145/1141277.1141564`.

[Van+03] **Van Der Straeten, R., Mens, T., Simmonds, J., and Jonckers, V.** "Using Description Logic to Maintain Consistency between UML Models". In: *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications.* Ed. by **Stevens, P., Whittle, J., and Booch, G.** Vol. 2863. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2003, pp. 326–340. DOI: `10.1007/978-3-540-45221-8_28`.

[VDI04] **VDI**. *Design Methodology for Mechatronic Systems.* VDI Guideline VDI 2206. 2004.

[VDI10] **VDI**. *Engineering of industrial plants – Evaluation and optimization – Part 2: Subject Processes.* VDI Guideline VDI 3695-2. 2010.

[Vog15] **Vogel-Heuser, B.** "Usability Experiments to Evaluate UML/SysML-Based Model Driven Software Engineering Notations for Logic Control in Manufacturing Automation". In: *Journal of Software Engineering and Applications*, vol. 11, no. 7 (2015), pp. 943–973. DOI: `10.4236/jsea.2014.711084`.

[Vog+14a] **Vogel-Heuser, B., Diedrich, C., Fay, A., Jeschke, S., Kowalewski, S., and Wollschläger, M.** "Challenges for Software Engineering in Automation". In: *Journal of Software Engineering and Applications*, vol. 7, no. 5 (2014). DOI: `10.4236/jsea.2014.75041`.

[VH16] **Vogel-Heuser, B. and Hess, D.** "Guest Editorial Industry 4.0 – Prerequisites and Visions". In: *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2 (2016), pp. 411–413. DOI: `10.1109/TASE.2016.2523639`.

[Vog+13] **Vogel-Heuser, B., Obermeier, M., Braun, S., Sommer, K., Jobst, F., and Schweizer, K.** "Evaluation of a UML-Based Versus an IEC 61131-3-Based Software Engineering Approach for Teaching PLC Programming". In: *IEEE Transactions on Education*, vol. 56, no. 3 (2013), pp. 329–335. DOI: `10.1109/TE.2012.2226035`.

[VR15] **Vogel-Heuser, B. and Rösch, S.** "Applicability of Technical Debt as a Concept to Understand Obstacles for Evolution of Automated Production Systems". In: *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on.* 2015, pp. 127–132. DOI: `10.1109/SMC.2015.35`.

[Vog+15] **Vogel-Heuser, B., Fay, A., Schaefer, I., and Tichy, M.** "Evolution of Software in Automated Production Systems: Challenges and Research Directions". In: *Journal of Systems and Software*, vol. 110 (2015). DOI: `10.1016/j.jss.2015.08.026`.

[Vog+14b] **Vogel-Heuser, B., Legat, C., Folmer, J., and Feldmann, S.** *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit.* Ed. by **Institute of Automation and Information Systems**. Online. 2014. URL: `https://mediatum.ub.tum.de/node?id=1208973` (visited on 04/10/2016).

[Vya11] **Vyatkin, V.** "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-art Review". In: *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4 (2011), pp. 768–781. DOI: `10.1109/TII.2011.2166785`.

[W3C12] **W3C**. *Web Ontology Language (OWL) 2 – Structural Specification and Function-Style Syntax.* 2012. URL: `http://www.w3.org/TR/owl-syntax` (visited on 04/10/2016).

[W3C13a] **W3C**. *SPARQL Protocol and RDF Query Language (SPARQL) 1.1 Query Language.* 2013. URL: `https://www.w3.org/TR/sparql11-query/` (visited on 04/10/2016).

[W3C13b] **W3C**. *SPARQL Protocol and RDF Query Language (SPARQL) 1.1 Update.* 2013. URL: `https://www.w3.org/TR/sparql11-update/` (visited on 04/10/2016).

[W3C14a] **W3C**. *Mathematical Markup Language (MathML) Version 3.0.* 2014. URL: `https://www.w3.org/TR/MathML3/` (visited on 04/10/2016).

[W3C14b] **W3C**. *Resource Description Framework (RDF) 1.1 Concepts and Abstract Syntax.* 2014. URL: https://www.w3.org/TR/rdf11-concepts/ (visited on 04/10/2016).

[W3C14c] **W3C**. *Resource Description Framework (RDF) 1.1 Turtle – Terse RDF Triple Language.* 2014. URL: https://www.w3.org/TR/turtle/ (visited on 04/10/2016).

[W3C14d] **W3C**. *Resource Description Framework (RDF) Schema 1.1.* 2014. URL: https://www.w3.org/TR/rdf-schema/ (visited on 04/10/2016).

[Wer09] **Werner, B.** "Object-oriented Extensions for IEC 61131-3". In: *IEEE Industrial Electronics Magazine*, vol. 3, no. 4 (2009), pp. 36–39. DOI: 10.1109/MIE.2009.934795.

[WB12] **Winkler, D. and Biffl, S.** "Improving Quality Assurance in Automation Systems Development Projects". In: InTech, 2012. DOI: 10.5772/33487.

[WB15] **Winkler, D. and Biffl, S.** "Focused Inspections to Support Defect Detection in Automation Systems Engineering Environments". In: *International Conference on Product-Focused Software Process Improvement.* 2015, pp. 372–379. DOI: 10.1007/978-3-319-26844-6_27.

[WV11] **Witsch, D. and Vogel-Heuser, B.** "PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking". In: *IFAC World Congress.* Milano, Italy, 2011. DOI: 10.3182/20110828-6-IT-1002.02207.

[Wol19] **Wolfenstetter, T.** "Model Integration and Traceability for Product Service Systems Engineering". PhD thesis. Technische Universität München, 2019.

[Wol+18] **Wolfenstetter, T., Basirati, M. R., Böhm, M., and Krcmar, H.** "Introducing TRAILS: A tool supporting traceability, integration and visualisation of engineering knowledge for product service systems development". In: *Journal of Systems and Software*, vol. 144 (2018), pp. 342–355. DOI: 10.1016/j.jss.2018.06.079.

[Zou+19] **Zou, M., Basirati, M., Bauer, H., Kattner, N., Reinhart, G., Lindemann, U., Böhm, M., Krcmar, H., and Vogel-Heuser, B.** "Facilitating Consistency of Business Model and Technical Models in Product-Service- Systems Development: An Ontology Approach". In: *IFAC Conference on Manufacturing Modelling, Management and Control.* 2019.

[ZLV18] **Zou, M., Lu, B., and Vogel-Heuser, B.** "Resolving Inconsistencies Optimally in the Model-Based Development of Production Systems". In: *IEEE International Conference on Automation Science and Engineering.* 2018, pp. 1064–1070.

[ZV17] **Zou, M. and Vogel-Heuser, B.** "Feature-based systematic approach development for inconsistency resolution in automated production system design". In: *IEEE Conference on Automation Science and Engineering.* 2017, pp. 687–694. DOI: 10.1109/COASE.2017.8256183.

List of Figures

List of Tables

List of Listings

List of Symbols and Abbreviations

ATL	ATLAS Transformation Language
AutomationML	Automation Markup Language
BOM	Bill of Material
BPMN	Business Process Model and Notation
CAD	Computer-Aided Design
CAEX	Computer Aided Engineering Exchange
COLLADA	Collaborative Design Activity
CRC	Collaborative Research Centre
DFG	Deutsche Forschungsgemeinschaft
DL	Description Logic
DSL	Domain-specific Language
EMF	Eclipse Modeling Framework
ERP	Enterprise Resource Planning
EVL	Epsilon Validation Language
FB	Function Block
FBD	Function Block Diagram
IL	Instruction List
INCOSE	International Council on Systems Engineering
LD	Ladder Diagram
MathML	Mathematical Markup Language
MBE	Model-Based Engineering
MBSE	Model-Based Systems Engineering
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta Object Facility
MOFM2T	MOF Model To Text
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
OCL	Object Constraint Language
OMG	Object Management Group
OWL	Web Ontology Language
PC	Personal Computer

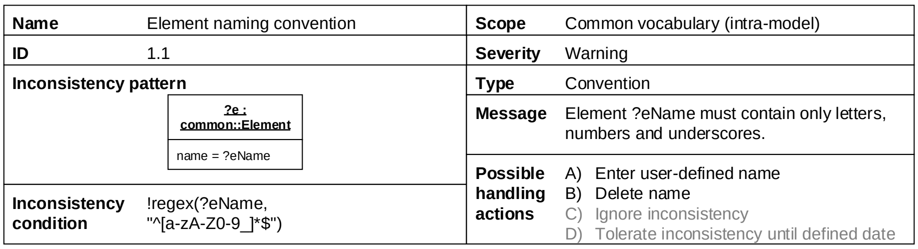
PLC	Programmable Logic Controller
PLM	Product Lifecycle Management
POU	Program Organization Unit
PPU	Pick and Place Unit
QUDT	Quantity, Unit, Dimension and Type
RDF	Resource Description Framework
RDFS	RDF Schema
SFC	Sequential Function Chart
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SQWRL	Semantic Query-Enhanced Web Rule Language
ST	Structured Text
SWRL	Semantic Web Rule Language
SysML	Systems Modeling Language
TGG	Triple Graph Grammar
UML	Unified Modeling Language
UNA	Unique Name Assumption
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WP	work piece
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformation

Appendix A.

Inconsistencies

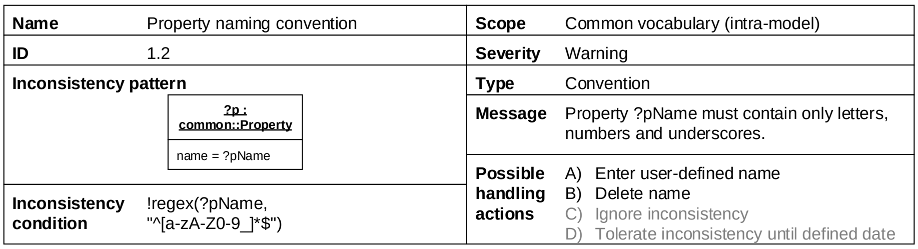
A.1. Inconsistencies for Evaluation Stage 1

A.1.1. Intra-model Inconsistencies



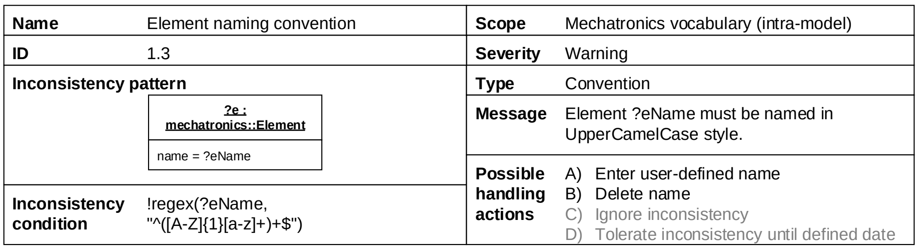
Name	Element naming convention	**Scope**	Common vocabulary (intra-model)
ID	1.1	**Severity**	Warning
Inconsistency pattern	?e : common::Element name = ?eName	**Type**	Convention
		Message	Element ?eName must contain only letters, numbers and underscores.
		Possible handling actions	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	!regex(?eName, "^[a-zA-Z0-9_]*$")		

Figure A.1. Inconsistency 1.1: Naming inconsistency for elements (common vocabulary)



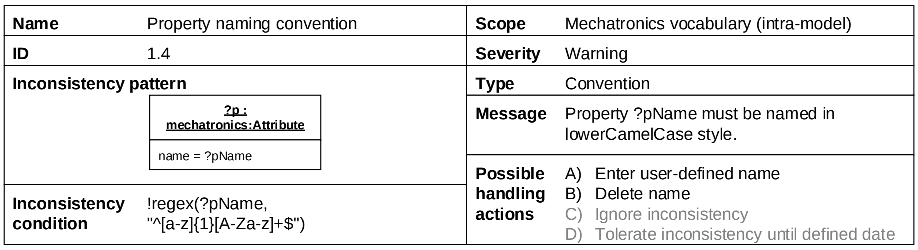
Name	Property naming convention	**Scope**	Common vocabulary (intra-model)
ID	1.2	**Severity**	Warning
Inconsistency pattern	?p : common::Property name = ?pName	**Type**	Convention
		Message	Property ?pName must contain only letters, numbers and underscores.
		Possible handling actions	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	!regex(?pName, "^[a-zA-Z0-9_]*$")		

Figure A.2. Inconsistency 1.2: Naming inconsistency for properties (common vocabulary)



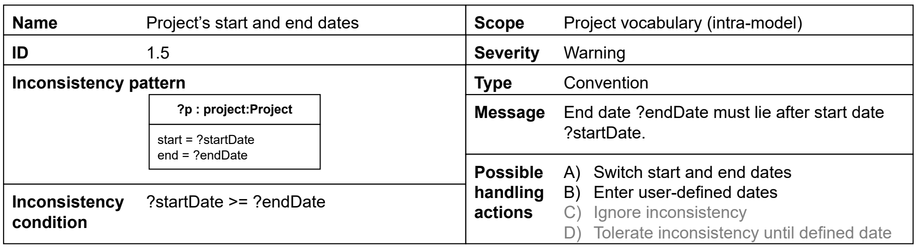
Name	Element naming convention	**Scope**	Mechatronics vocabulary (intra-model)
ID	1.3	**Severity**	Warning
Inconsistency pattern	?e : mechatronics::Element name = ?eName	**Type**	Convention
		Message	Element ?eName must be named in UpperCamelCase style.
		Possible handling actions	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	!regex(?eName, "^([A-Z]{1}[a-z]+)+$")		

Figure A.3. Inconsistency 1.3: Naming inconsistency for elements (mechatronics vocabulary)



Name	Property naming convention	**Scope**	Mechatronics vocabulary (intra-model)
ID	1.4	**Severity**	Warning
Inconsistency pattern	?p : mechatronics:Attribute name = ?pName	**Type**	Convention
		Message	Property ?pName must be named in lowerCamelCase style.
		Possible handling actions	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	!regex(?pName, "^[a-z]{1}[A-Za-z]+$")		

Figure A.4. Inconsistency 1.4: Naming inconsistency for properties (mechatronics vocabulary)

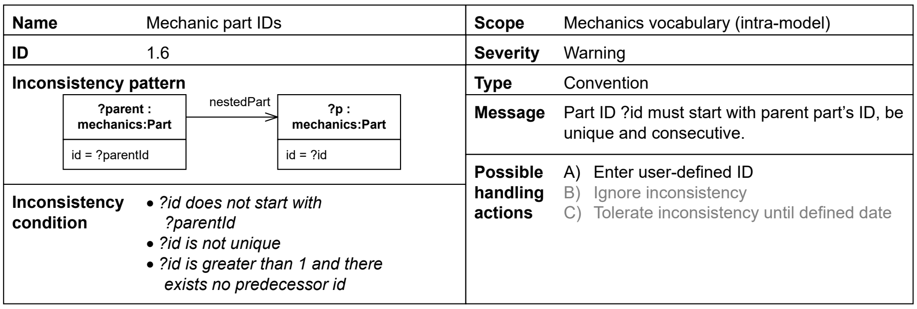


Name	Project's start and end dates	**Scope**	Project vocabulary (intra-model)
ID	1.5	**Severity**	Warning

Inconsistency pattern	**?p : project:Project**
	start = ?startDate
	end = ?endDate
		Type	Convention

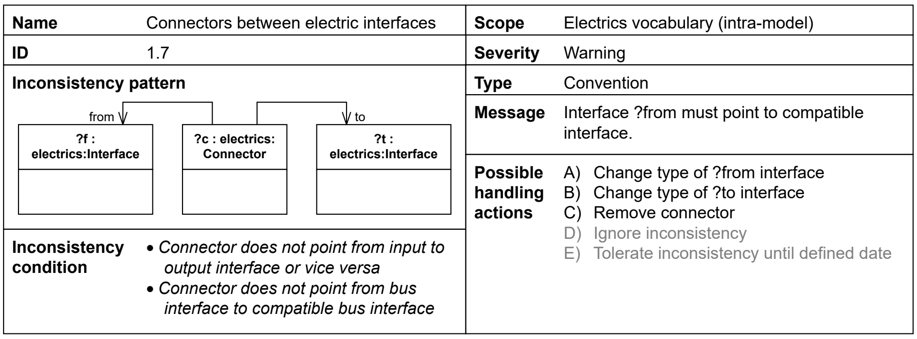
		Message	End date ?endDate must lie after start date ?startDate.
Inconsistency condition	?startDate >= ?endDate	**Possible handling actions**	A) Switch start and end dates B) Enter user-defined dates C) Ignore inconsistency D) Tolerate inconsistency until defined date

Figure A.5. Inconsistency 1.5: Inconsistency regarding project's start and end dates



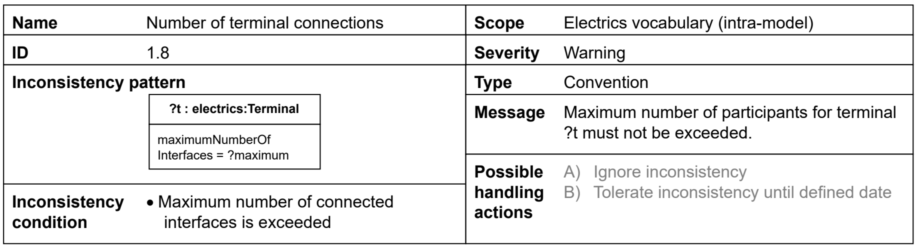
Name	Mechanic part IDs	**Scope**	Mechanics vocabulary (intra-model)
ID	1.6	**Severity**	Warning
Inconsistency pattern	**?parent : mechanics:Part** id = ?parentId nestedPart **?p : mechanics:Part** id = ?id	**Type**	Convention
		Message	Part ID ?id must start with parent part's ID, be unique and consecutive.
Inconsistency condition	• *?id does not start with ?parentId* • *?id is not unique* • *?id is greater than 1 and there exists no predecessor id*	**Possible handling actions**	A) Enter user-defined ID B) Ignore inconsistency C) Tolerate inconsistency until defined date

Figure A.6. Inconsistency 1.6: Inconsistency regarding mechanic parts' IDs



Name	Connectors between electric interfaces	**Scope**	Electrics vocabulary (intra-model)
ID	1.7	**Severity**	Warning
Inconsistency pattern	from to **?f : electrics:Interface** **?c : electrics: Connector** **?t : electrics:Interface**	**Type**	Convention
		Message	Interface ?from must point to compatible interface.
Inconsistency condition	• *Connector does not point from input to output interface or vice versa* • *Connector does not point from bus interface to compatible bus interface*	**Possible handling actions**	A) Change type of ?from interface B) Change type of ?to interface C) Remove connector D) Ignore inconsistency E) Tolerate inconsistency until defined date

Figure A.7. Inconsistency 1.7: Inconsistency regarding electrical interfaces



Name	Number of terminal connections	**Scope**	Electrics vocabulary (intra-model)
ID	1.8	**Severity**	Warning
Inconsistency pattern	**?t : electrics:Terminal** maximumNumberOf Interfaces = ?maximum	**Type**	Convention
		Message	Maximum number of participants for terminal ?t must not be exceeded.
Inconsistency condition	• Maximum number of connected interfaces is exceeded	**Possible handling actions**	A) Ignore inconsistency B) Tolerate inconsistency until defined date

Figure A.8. Inconsistency 1.8: Inconsistency regarding terminals and bus couplers (shown at the example of terminals)

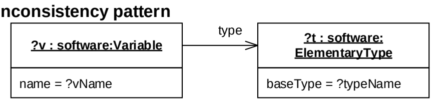
Name	Software variable names	**Scope**	Software vocabulary (intra-model)
ID	1.9	**Severity**	Warning
Inconsistency pattern  ?v : software:Variable name = ?vName type ?t : software: ElementaryType baseType = ?typeName		**Type**	Convention
		Message	Software variable ?vName must follow the Hungarian notation.
Inconsistency condition	(?typeName = "INT" && !regex(?vName, "^i([A-Z]{1}[a-z]+)+$")) \|\| (?typeName = "BOOL" && !regex(?vName, "^b([A-Z]{1}[a-z]+)+$")) || ...	**Possible handling actions**	A) Auto-generate exemplary name B) Enter user-defined name C) Ignore inconsistency D) Tolerate inconsistency until defined date

Figure A.9. Inconsistency 1.9: Software variable naming conventions

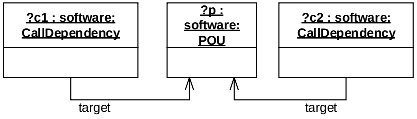
Name	Call tree	**Scope**	Software vocabulary (intra-model)
ID	1.10	**Severity**	Warning
Inconsistency pattern  ?c1 : software: CallDependency ?p : software: POU ?c2 : software: CallDependency target target		**Type**	Convention
		Message	?p is called twice – hence, the call tree is violated.
Inconsistency condition	• ?c1 != ?c2	**Possible handling actions**	A) Ignore inconsistency B) Tolerate inconsistency until defined date

Figure A.10. Inconsistency 1.10: Inconsistency regarding software call hierarchy

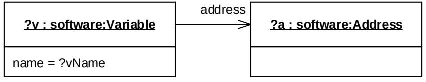
Name	Global variables with addresses	**Scope**	Software vocabulary (intra-model)
ID	1.11	**Severity**	Warning
Inconsistency pattern  ?v : software:Variable name = ?vName address ?a : software:Address		**Type**	Convention
		Message	Software variable ?v must be read or written at least once
Inconsistency condition	• *Global variables with an assigned address is neither read nor written*	**Possible handling actions**	A) Ignore inconsistency B) Tolerate inconsistency until defined date

Figure A.11. Inconsistency 1.11: Inconsistency regarding reading and writing global variables

A.1.2. Inter-model Inconsistencies

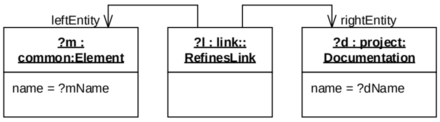
Name	Refinement between project documentation and discipline model	**Scope**	Project and discipline vocabularies (inter-model)
ID	1.12	**Severity**	Warning
Inconsistency pattern  leftEntity rightEntity ?m : common:Element name = ?mName ?l : link:: RefinesLink ?d : project: Documentation name = ?dName		**Type**	Correspondence
		Message	Project documentation ?dName must be refined by a discipline model
Inconsistency condition	• ?mName != ?dName • *?m is not a mechanic part list, an electric circuit model or a software dependency model*	**Possible handling actions**	A) Create discipline model with according link B) Propagate name from left to right C) Propagate name from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date

Figure A.12. Inconsistency 1.12: Inconsistency for refinement between project documentation and discipline-specific model

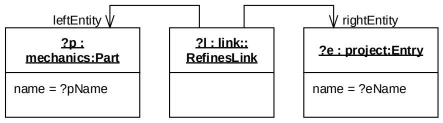
Name	Refinement between entry in BOM and mechanic parts	**Scope**	Project and mechanics vocabularies (inter-model)
ID	1.13	**Severity**	Warning
Inconsistency pattern	 leftEntity rightEntity ?p : mechanics:Part name = ?pName ?l : link:: RefinesLink ?e : project:Entry name = ?eName	**Type**	Correspondence
		Message	BOM entry ?eName must be refined by a respective mechanic part.
Inconsistency condition	• ?pName != ?dName	**Possible handling actions**	A) Create refining part with according link B) Propagate name from left to right C) Propagate name from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date

Figure A.13. Inconsistency 1.13: Inconsistency for refinement between mechanical part and entry in the project's BOM

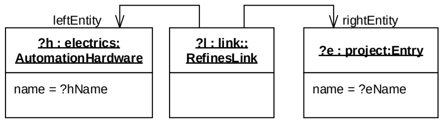
Name	Refinement between entry in BOM and electric automation hardware	**Scope**	Project and electrics vocabularies (inter-model)
ID	1.14	**Severity**	Warning
Inconsistency pattern	 leftEntity rightEntity ?h : electrics: AutomationHardware name = ?hName ?l : link:: RefinesLink ?e : project:Entry name = ?eName	**Type**	Correspondence
		Message	BOM entry ?eName must be refined by a respective electric part.
Inconsistency condition	• ?hName != ?dName	**Possible handling actions**	A) Create refining part with according link B) Propagate name from left to right C) Propagate name from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date

Figure A.14. Inconsistency 1.14: Inconsistency for refinement between electrical automation hardware and entry in the project's BOM

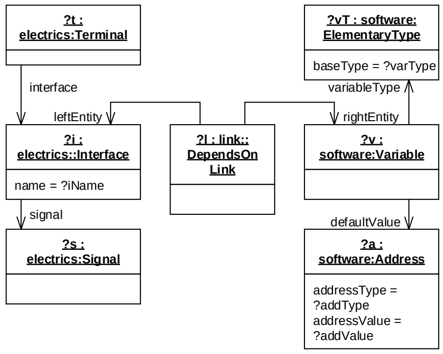
Name	Dependence between electric interface and software variable	**Scope**	Electrics and software vocabularies (inter-model)
ID	1.15	**Severity**	Warning
Inconsistency pattern	 ?t : electrics:Terminal ?vT : software: ElementaryType baseType = ?varType interface variableType leftEntity rightEntity ?i : electrics::Interface name = ?iName ?l : link:: DependsOn Link ?v : software:Variable signal defaultValue ?s : electrics:Signal ?a : software:Address addressType = ?addType addressValue = ?addValue	**Type**	Correspondence
		Message	Interface ?i must be consistent to linked software variable ?v.
		Possible handling actions	A) Create software variable with according link B) Ignore inconsistency C) Tolerate inconsistency until defined date
Inconsistency condition	• *Software variable is not a global variable* • *Software address defines input, but electric interface is output* • *Software address defines output, but electric interface is input* • *Address value does not match interface* • *Software variable type does not match signal type*		

Figure A.15. Inconsistency 1.15: Inconsistency for dependency between terminal interface and software variable

A.2. Inconsistencies for Evaluation Stage 2

Name	Matrix-based device degradation classification	**Scope**	Project and degradation vocabularies (inter-model)
ID	2.1	**Severity**	Error
Type	Correspondence	**Message**	Device ?h (order number ?oNr) should not be used under current environment conditions.
Inconsistency condition	?humidity >= ?criticalHum || ?temperature >= ?criticalTemp		

Inconsistency pattern

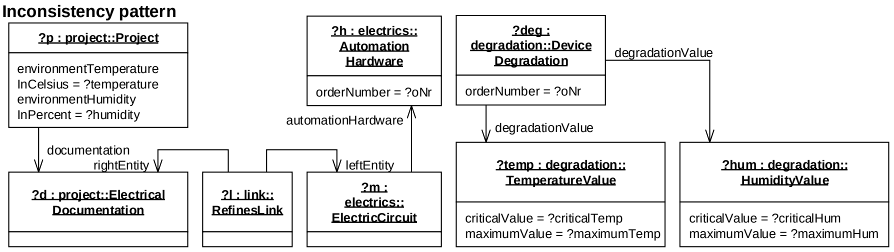

?p : project::Project
environmentTemperature InCelsius = ?temperature
environmentHumidity InPercent = ?humidity
?h : electrics:: Automation Hardware
orderNumber = ?oNr
?deg : degradation::Device Degradation
orderNumber = ?oNr
degradationValue
automationHardware
degradationValue
documentation
rightEntity
leftEntity
?d : project::Electrical Documentation
?l : link:: RefinesLink
?m : electrics:: ElectricCircuit
?temp : degradation:: TemperatureValue
criticalValue = ?criticalTemp
maximumValue = ?maximumTemp
?hum : degradation:: HumidityValue
criticalValue = ?criticalHum
maximumValue = ?maximumHum

Figure A.16. Inconsistency 2.1: Inconsistency for device degradation

Name	Linear coupler degradation	**Scope**	Project and degradation vocabularies (inter-model)
ID	2.2a	**Severity**	Error
Type	Correspondence	**Message**	Coupler ?c has ?actual connections, but must not exceed maximum connections under environment conditions.
Inconsistency condition	?actual > (?max * (1 – 1 / ?critTMax * ?temp))		

Inconsistency pattern

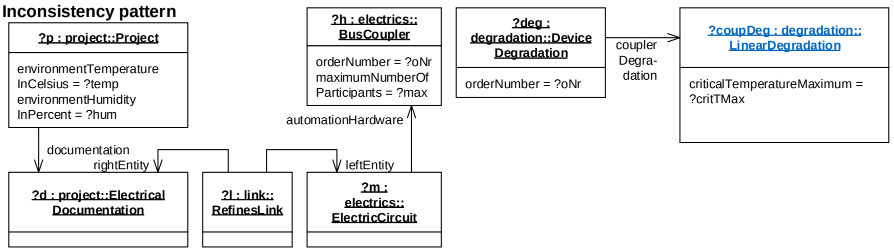

?p : project::Project
environmentTemperature InCelsius = ?temp
environmentHumidity InPercent = ?hum
?h : electrics:: BusCoupler
orderNumber = ?oNr
maximumNumberOf Participants = ?max
?deg : degradation::Device Degradation
orderNumber = ?oNr
coupler Degra- dation
?coupDeg : degradation:: LinearDegradation
criticalTemperatureMaximum = ?critTMax
automationHardware
documentation
rightEntity
leftEntity
?d : project::Electrical Documentation
?l : link:: RefinesLink
?m : electrics:: ElectricCircuit

Figure A.17. Inconsistency 2.2a: Inconsistency for linear coupler degradation

Name	Cubic coupler degradation	**Scope**	Project and degradation vocabularies (inter-model)
ID	2.2b	**Severity**	Error
Type	Correspondence	**Message**	Coupler ?c has ?actual connections, but must not exceed maximum connections under environment conditions.
Inconsistency condition	?actual > (?max * (1 – 1 / ?critTMax^2 * ?temp^2))		

Inconsistency pattern

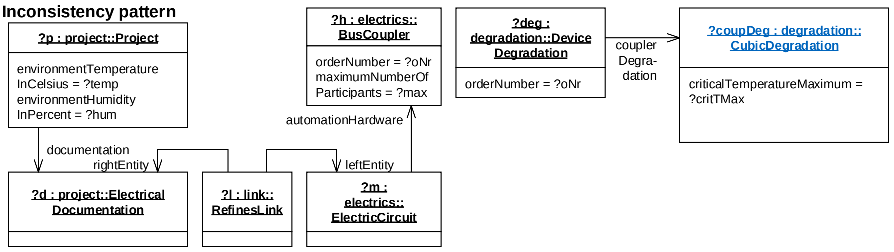
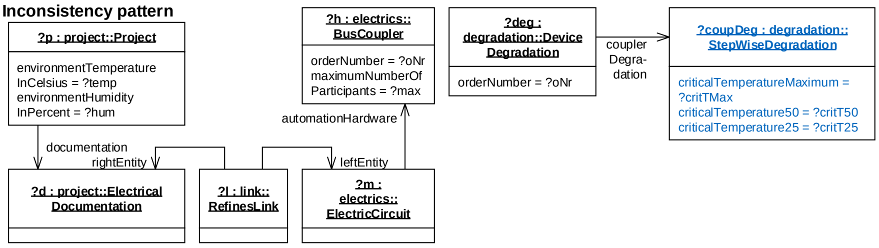

?p : project::Project
environmentTemperature InCelsius = ?temp
environmentHumidity InPercent = ?hum
?h : electrics:: BusCoupler
orderNumber = ?oNr
maximumNumberOf Participants = ?max
?deg : degradation::Device Degradation
orderNumber = ?oNr
coupler Degra- dation
?coupDeg : degradation:: CubicDegradation
criticalTemperatureMaximum = ?critTMax
automationHardware
documentation
rightEntity
leftEntity
?d : project::Electrical Documentation
?l : link:: RefinesLink
?m : electrics:: ElectricCircuit

Figure A.18. Inconsistency 2.2b: Inconsistency for cubic coupler degradation

Name	Step-wise coupler degradation	**Scope**	Project and degradation vocabularies (inter-model)
ID	2.2c	**Severity**	Error
Type	Correspondence	**Message**	Coupler ?c has ?actual connections, but must not exceed maximum connections under environment conditions.
Inconsistency condition	?actual > (**IF** (?temp > ?critTmax) **THEN** 0 **ELSE IF** (?temp > ?critT25) **THEN** 0.25*?max **ELSE IF** (?temp > ?critT50) **THEN** 0.5*?max)		
Inconsistency pattern			

?p : project::Project
environmentTemperature InCelsius = ?temp
environmentHumidity InPercent = ?hum
?h : electrics:: BusCoupler
orderNumber = ?oNr
maximumNumberOf Participants = ?max
?deg : degradation::Device Degradation
orderNumber = ?oNr
coupler Degra- dation
?coupDeg : degradation:: StepWiseDegradation
criticalTemperatureMaximum = ?critTMax
criticalTemperature50 = ?critT50
criticalTemperature25 = ?critT25
automationHardware
documentation
rightEntity
leftEntity
?d : project::Electrical Documentation
?l : link:: RefinesLink
?m : electrics:: ElectricCircuit

Figure A.19. Inconsistency 2.2c: Inconsistency for step-wise coupler degradation

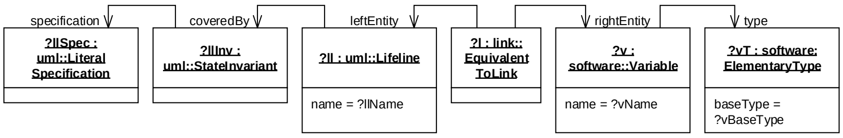
Name	Equivalence between UML Lifeline and Software Variable	**Scope**	UML and software vocabularies (inter-model)
ID	2.3	**Severity**	Warning
Type	Correspondence	**Message**	Software Variable ?vName should be further specified by respective consistent UML Lifeline.
Inconsistency condition	?llName != ?vName || !(?vBaseType = "BOOL" && LiteralBoolean(?llSpec)) || !(?vBaseType = "INT" && LiteralInteger(?llSpec))		
Inconsistency pattern			

Figure A.20. Inconsistency 2.3: Inconsistency for equivalence between UML Lifeline and software Variable

A.3. Inconsistencies for Evaluation Stage 3

A.3.1. Intra-model Inconsistencies

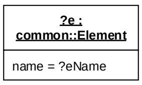
Name	Element naming convention	**Scope**	Common vocabulary (intra-model)
ID	3.1	**Severity**	Warning
Inconsistency pattern		**Type**	Convention
		Message	Element ?eName must be named in UpperCamelCase style.
		Possible handling actions	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	!regex(?eName, "^([A-Z]{1}[a-z]+)+$")		

Figure A.21. Inconsistency 3.1: Naming inconsistency for elements

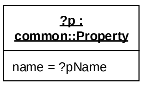
Name	Property naming convention	**Scope**	Common vocabulary (intra-model)
ID	3.2	**Severity**	Warning
Inconsistency pattern 		**Type**	Convention
		Message	Property ?pName must be named in lowerCamelCase style.
Inconsistency condition	!regex(?pName, "^[a-z]{1}[A-Za-z]+$")	**Possible handling actions**	A) Enter user-defined name B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date

Figure A.22. Inconsistency 3.2: Naming inconsistency for properties

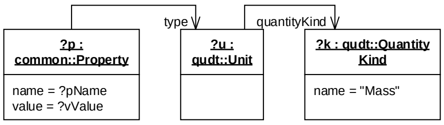
Name	Negative mass values	**Scope**	Common vocabulary (intra-model)
ID	3.3	**Severity**	Error
Inconsistency pattern 		**Type**	Domain-specific
		Message	Mass property ?p must not be negative (value is ?value).
		Possible handling actions	A) Enter user-defined value for property B) Delete name C) Ignore inconsistency D) Tolerate inconsistency until defined date
Inconsistency condition	?value < 0		

Figure A.23. Inconsistency 3.3: Negative mass value inconsistency

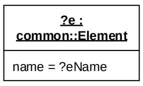
Name	Maximum hierarchy level	**Scope**	Common vocabulary (intra-model)
ID	3.4	**Severity**	Warning
Inconsistency pattern 		**Type**	Convention
		Message	Element ?e has 4 or more transitive contains relationships – maximum hierarchy level exceeded.
Inconsistency condition	*?e has 4 or more transitive contains relations*	**Possible handling actions**	A) Ignore inconsistency B) Tolerate inconsistency until defined date

Figure A.24. Inconsistency 3.4: Maximum hierarchy level inconsistency

A.3.2. Inter-model Inconsistencies

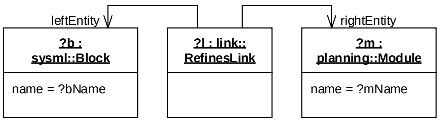
Name	Refinement between SysML Blocks and Planning Modules	**Scope**	Planning and SysML vocabularies (inter-model)
ID	3.5	**Severity**	Warning
Inconsistency pattern 		**Type**	Correspondence
		Message	Module ?m must be refined by equivalently named Block ?b.
		Possible handling actions	A) Create refining block with according link B) Propagate name from left to right C) Propagate name from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date
Inconsistency condition	?mName != ?bName		

Figure A.25. Inconsistency 3.5: Inconsistency for refinement between SysML Block and Planning Module

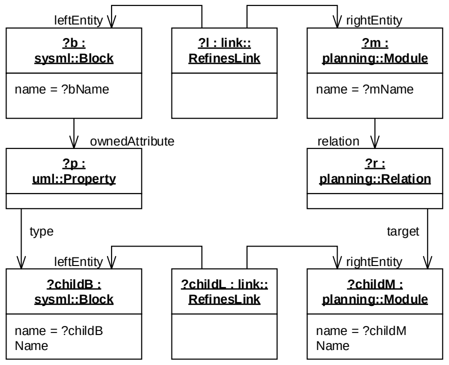
Name	Refinement between SysML Child Blocks and Planning Child Modules	**Scope**	Planning and SysML vocabularies (inter-model)
ID	3.6	**Severity**	Warning
Inconsistency pattern		**Type**	Correspondence
		Message	Child Module ?childM must be refined by equivalently named child Block ?childB.
		Possible handling actions	A) Create refining block with according link B) Propagate name from left to right C) Propagate name from right to left D) Ignore inconsistency E) Tolerate inconsistency until defined date
Inconsistency condition	?childMName != ?childBName		

Figure A.26. Inconsistency 3.6: Inconsistency for refinement between SysML Block hierarchy and Planning Module hierarchy

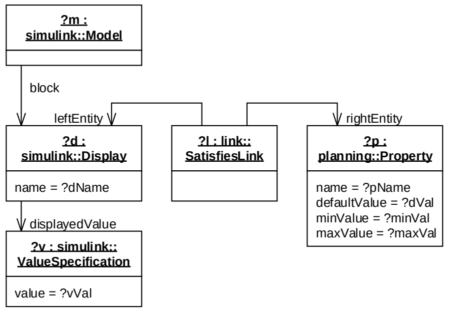
Name	Satisfaction between top-level Simulink Display and Planning Property	**Scope**	Planning and Simulink vocabularies (inter-model)
ID	3.7	**Severity**	Warning
Inconsistency pattern		**Type**	Correspondence
		Message	Display ?d (value ?vVal) must satisfy respective Property ?p (defaultValue ?dVal, minValue ?minVal, maxValue ?maxVal).
		Possible handling actions	A) Create property with according link B) Ignore inconsistency C) Tolerate inconsistency until defined date
Inconsistency condition	?vVal < ?minVal || ?vVal > ?maxVal || ?vVal != ?dVal		

Figure A.27. Inconsistency 3.7: Inconsistency for satisfaction between top-level MATLAB/Simulink Display and Planning Property

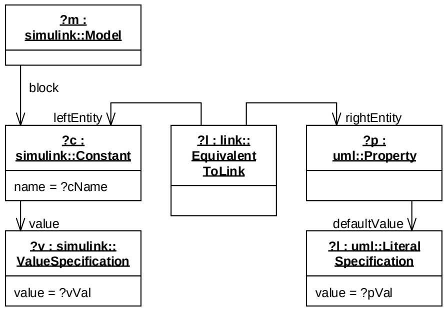
Name	Equivalence between top-level Simulink Constant and SysML Property	**Scope**	SysML and Simulink vocabularies (inter-model)
ID	3.8	**Severity**	Warning
Inconsistency pattern  ?m : simulink::Model block leftEntity rightEntity ?c : simulink::Constant name = ?cName ?l : link:: Equivalent ToLink ?p : uml::Property value defaultValue ?v : simulink:: ValueSpecification value = ?vVal ?l : uml::Literal Specification value = ?pVal		**Type**	Correspondence
		Message	Constant ?c (value ?cVal) must be equivalent to respective Property ?p (defaultValue ?pVal).
		Possible handling actions	A) Create property with according link B) Ignore inconsistency C) Tolerate inconsistency until defined date
Inconsistency condition	?vVal != ?pVal		

Figure A.28. Inconsistency 3.8: Inconsistency for equivalence between top-level MATLAB/Simulink Constant and UML Property