# Chapter 14
# Applications of Semantic Web Technologies for the Engineering of Automated Production Systems—Three Use Cases

**Stefan Feldmann, Konstantin Kernschmidt and Birgit Vogel-Heuser**

**Abstract** The increasing necessity to adapt automated production systems rapidly to changing requirements requires a better support for planning, developing and operating automated production systems. One means to improve the engineering of these complex systems is the use of models, thereby abstracting the view on the system and providing a common base to improve understanding and communication between engineers. However, in order for any engineering project to be successful, it is essential to keep the created engineering models consistent. We envision the use of Semantic Web Technologies for such consistency checks in the domain of Model-Based Engineering. In this chapter, we show how Semantic Web Technologies can support consistency checking for the engineering process in the automated production systems domain through three distinct use cases: In a first use case, we illustrate the combination of a Systems Modeling Language-based notation with Web Ontology Language (OWL) to ensure compatibility between mechatronic modules after a module change. A second use case demonstrates the application of OWL with the SPARQL Query Language to ensure consistency during model-based requirements and test case design for automated production systems. In a third use case, it is shown how the combination of the Resource Description Framework (RDF) and the SPARQL Query Language can be used to identify inconsistencies between interdisciplinary engineering models of automated production systems. We conclude with opportunities of applying Semantic Web Technologies to support the engineering of automated production systems and derive the research questions that need to be answered in future work.

**Keywords** Automated production systems · Semantic Web Technologies · Model-Based Engineering · Knowledge-based systems · Inconsistency management

S. Feldmann (✉) · K. Kernschmidt · B. Vogel-Heuser
Institute of Automation and Information Systems, Technische Universität München,
Boltzmannstraße 15, 85748 Garching near Munich, Germany
e-mail: feldmann@ais.mw.tum.de

K. Kernschmidt
e-mail: kernschmidt@ais.mw.tum.de

B. Vogel-Heuser
e-mail: vogel-heuser@ais.mw.tum.de

© Springer International Publishing Switzerland 2016

S. Biffl and M. Sabou (eds.), *Semantic Web Technologies for Intelligent Engineering Applications*, DOI 10.1007/978-3-319-41490-4_14

## 14.1 Introduction

Rapidly evolving system and product requirements impose an increasing need regarding the efficiency and effectivity in engineering of automated production systems. Whereas such systems were mainly dominated through mechanical and electrical/electronic parts within the last decades, the significance of software is and will be increasing (Strasser et al. 2009). As a consequence, automated production systems need to fulfill ever increasing functionality and, thus, are becoming more and more complex. It is obvious that the engineering of automated production systems must follow this trend: methods and tools to better support engineers in developing and operating automated production systems need to be developed and further improved.

One means to improve the engineering of automated production systems is the use of Model-Based Engineering, thereby abstracting the view on the interdisciplinary system and providing a common base to improve understanding and communication. However, the multitude of disciplines and persons involved in the engineering process of automated production systems requires the use of a variety of different modeling languages, formalisms, and levels of abstraction and, hence, a number of disparate, but mostly overlapping models is created during engineering. Therefore, there is a need for tool support for, e.g., finding model elements within the models and keeping the engineering models consistent among each other.

In order to overcome this challenge, this Chapter illustrates how applying Semantic Web Technologies can support Model-Based Engineering activities in the automated production systems domain. Based on an application example (Sect. 14.2), the challenges that arise during engineering of automated production systems are presented in Sect. 14.3, followed by a short introduction of related works in the field of inconsistency management in Sect. 14.4. Subsequently, the technologies being used throughout this Chapter, namely, Resource Description Framework (RDF), Web Ontology Language (OWL), and SPARQL Query Language, are briefly introduced in Sect. 14.5. Three distinct use cases that illustrate how Semantic Web Technologies can support the engineering are introduced in Sect. 14.6:

- *Use case 1* (Sect. 14.6.1) describes the combination of a Systems Modeling Language (SysML)-based notation with OWL to ensure compatibility between mechatronic modules after a change of modules.
- *Use case 2* (Sect. 14.6.2) demonstrates how consistency can be ensured during model-based requirements and test case design by means of OWL and the SPARQL Query Language.
- In *use case 3* (Sect. 14.6.3) we show how the combination of RDF and the SPARQL Query Language can be used to identify inconsistencies between interdisciplinary engineering models of automated production systems.

The paper closes with a summary of the opportunities that can be gained for engineering of automated production systems by means of Semantic Web Technologies and with directions for future research.

## 14.2 Application Example: The Pick and Place Unit

In the following, as a basis to illustrate our research results, we introduce an application example, namely, the *Pick and Place Unit (PPU)*. The PPU is a bench-scale academic demonstration case derived from industrial use cases to evaluate research approaches and results at different stages of the life cycle in the automated manufacturing systems domain. Although the PPU is a simple demonstration case, it is complex enough to demonstrate an excerpt of the challenges that arise during engineering and operation of automated production system. To provide an evaluation environment as close to reality as possible, the PPU solely consists of industrial components. Furthermore, all scenarios and models that have been developed for the PPU were derived from real industrial use cases (Vogel-Heuser et al. 2014).

We assume that, in its initial configuration (Fig. 14.1), the PPU consists of four modules: a *stack*, a *crane*, a *stamp*, and a *ramp* module. The source of work pieces is represented through the stack module. Work pieces are pushed from the stack into a handover position, at which sensors are installed to identify the type of work pieces provided by the stack. Subsequently, depending on the type of work piece—either plastic or metallic—work pieces are transported to the stamp or ramp modules. In the stamp module, work pieces are detected at the handover position, subsequently positioned and clamped below a bistable cylinder, and finally stamped by applying pressure to the work piece. The final work piece depot is represented through the ramp module.

The transport between the different modules is realized by the crane module. Therein, a vacuum gripper, which is mounted on a boom, grips, or releases work

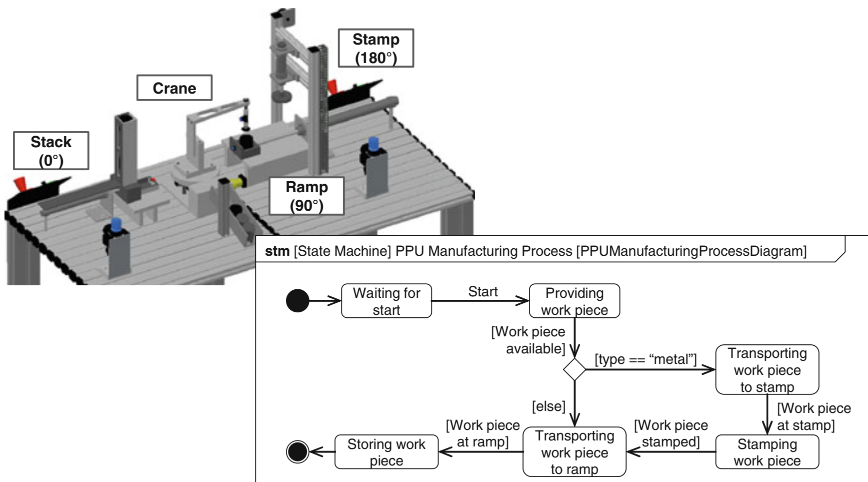



**Fig. 14.1** Overview on the Pick and Place Unit (PPU) and its manufacturing process, extended from Feldmann et al. (2015a)

pieces. Using a bistable cylinder, the boom and, hence, the work pieces held by the vacuum gripper are lifted or lowered. The entire assembly is mounted on a turning table. A DC motor attached to the turning table allows for moving the crane to the respective modules. In order to detect the current angular position of the crane, three micro switches are attached to the bottom of the turning table: one at 0° (stack), one at 90° (ramp), and one at 180° (stamp). Figure 14.2 provides a detailed overview of the modules of the PPU.

During the engineering of the PPU, a multitude of different engineering models is created. For instance, disparate discipline-specific models are used by different engineers to cover aspects from the mechanical, electrical/electronic, or software engineering discipline (cf. use case 1 in Sect. 14.6.1) as well as to capture requirements and generate test cases from these (cf. use case 2 in Sect. 14.6.2). All these disciplines require distinct modeling approaches, formalisms as well as tools in order to cover the aspects of interest for the respective discipline. Although one common

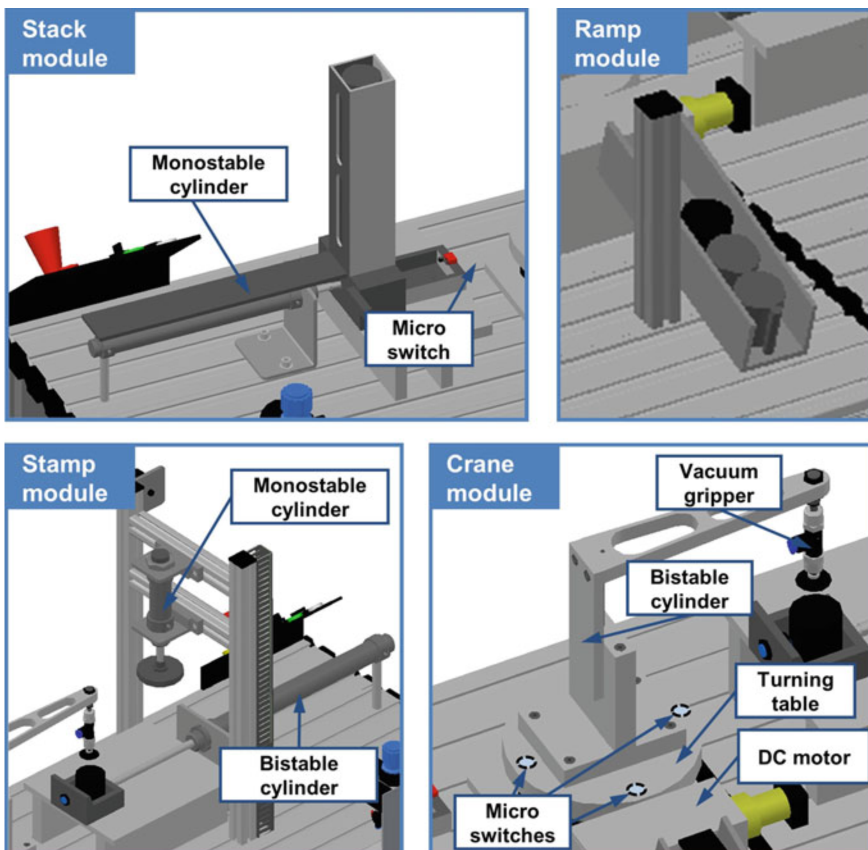



**Fig. 14.2** Modules of the Pick and Place Unit (PPU)

principle in Model-Based Engineering is to separate the resulting views on the system as much as possible, complete separation cannot be achieved, leading to overlaps in the models. One example for such an information overlap is the representation of components in the different disciplines: For instance, a sensor is represented as a geometrical block in a CAD system, as a digital input in an electrical circuit diagram, and as a respective software variable in the control software. As a consequence of creating these interdisciplinary engineering models, the risk of introducing inconsistencies arises, which makes appropriate strategies for identifying and resolving inconsistencies necessary (cf. use case 3 in Sect. 14.6.3).

## 14.3 Challenges in the Automated Production Systems Domain

Requirements on the system and product are changing rapidly in the automated production systems domain, especially in the context of Industrie 4.0 (cf. Chap. 2). As a consequence, a number of challenges arise that need to be addressed in the engineering phase of automated production systems. An excerpt of these challenges has been introduced in (Feldmann et al. 2015b) and is briefly described in the following:

**Challenge 1**: **Heterogeneity of Models**. In order to address the concerns of the various stakeholders involved in the design and development of automated production systems adequately, a multitude of different formalisms, modeling languages, and tools is required. As an example for the PPU demonstrator, a requirements model could be used for early phases of the design, whereas a detailed CAD model could be applied for the detailed planning of the mechanical plant layout. Moreover, while some models are needed to specify particular aspects of a system (e.g., the mechatronic structure of the PPU), others are applied for the purpose of analysis (e.g., to predict the work piece throughput of the PPU in a simulation model). As a consequence, a set of disparate models is created that make use of fundamentally different formalisms, different abstraction levels, and whose expressiveness is restricted to concepts that are relevant to a specific application domain. This heterogeneity of models poses a major challenge (Gausemeier et al. 2009), as the composition of these models is hard, if not impossible to define a priory. Consequently, mechanisms for symbolic manipulation across these heterogeneous models are required.

**Challenge 2**: **Semantic Overlaps Between Models**. Although heterogeneous models are created during the life cycle of automated production systems, these models are overlapping, as different stakeholders have overlapping views on the system under study. These overlaps result from the presence of either duplicate or related information. For instance, a requirement on the minimum work piece throughput could be imposed in an early design stage of the PPU. In later verification phases, a simulation model could be used to analyze the work piece throughput that can be predicted under the given circumstances and system configuration. Clearly, there is

a relation between the specified minimum work piece throughput and the predicted work piece throughout. Therefore, we say that statements are made about semantically related entities. We refer to such overlaps as *semantic overlaps*. As a consequence, such semantic overlaps between models need to be identified and specified as a basis for inconsistency management.

**Challenge 3**: **Automated Inconsistency Management within and Across Models**. Especially because of the heterogeneity of engineering models that are created for automated production systems (challenge 1) and the resulting overlap between these models (challenge 2), the risk of inconsistencies appearing in and across engineering models is high. Two strategies are commonly employed in order to manage these inconsistencies: (1) avoiding inconsistencies and (2) frequently searching for and handling inconsistencies. The first strategy requires to provide modelers with necessary and sufficient information about the different decisions, which are made by other stakeholders and impact their models. The second strategy is often employed in process models that include review (and testing) activities.

As models often consist of thousands of entities, at least some degree of automation and, hence, mechanisms for automated inconsistency management are required. Consequently, we argue that if a mechanism for symbolic manipulation across heterogeneous models is used and an appropriate method for identifying and defining semantic overlaps can be identified (i.e., if both challenges 1 and 2 can be addressed), both strategies become feasible. However, due to the lesser amount of knowledge that needs to be encoded and processed, we argue that the second strategy is likely to be more effective and less costly.

**Challenge 4**: **Integrated Tool Support**. In practice, especially in industrial environments, it is essential to keep the number of tools and methods that are applied as low as possible. In particular, for technicians and non-trained personnel, who are often involved during maintenance, start-up and operation of automated production systems, trainings are often costly. As a consequence, it is inevitable to integrate such support systems into existing tools instead of providing additional tools for the special purpose of managing inconsistencies. Thereby, stakeholders can work with the models and tools they are familiar with, without having to deal with additional models and tools focusing on inconsistency management. One essential challenge therein is to provide the mappings between the discipline-specific models and the respective symbolic manipulation mechanisms (cf. challenge 1) as a basis for inconsistency management.

## 14.4 Related Works in the Field of Inconsistency Management

In practice, inconsistency management is often included in complex review and verification tasks. However, the main challenge in identifying inconsistencies among

the multitude of heterogeneous engineering models is that tools and models are often loosely coupled (see, e.g., (Kovalenko et al. 2014) as well as Chaps. 8 and 13). Consequently, even if modeling tools can import and export models serialized in standardized formats such as XML (World Wide Web Consortium 2008) or AutomationML (International Electrotechnical Commission 2014), such tool-specific implementations may differ and, thus, lead to compatibility issues. In some cases, point-to-point integrations (e.g., using model transformations) and tool-specific wrappers are used to alleviate this challenge but, nevertheless, are fragile and costly to maintain (Feldmann et al. 2015b).

A comparison of inconsistency management approaches in the related literature (Feldmann et al. 2015a) revealed that these approaches can be broadly classified into *proof-theory-based* (i.e., deductive), *rule-based* (e.g., inductive and abductive) as well as model *synchronization-based* (i.e., model transformation-based) approaches (Feldmann et al. 2015a). In proof-theoretic approaches, consistency to an underlying formal system can be shown and, thus, proof-theoretic approaches provide logically correct results. However, they have several practical limitations. For instance, proofs of engineering models require a complete and consistent definition of the underlying formal system, which in most cases is labor-intensive (if even possible). Model synchronization-based approaches require the transformation between different formal systems, which is not always possible without encoding large amounts of additional knowledge and information in transformations. Due to the effort in creating these transformations, this is unlikely to be feasible (practically and economically). While being less formal than proof-theoretic approaches, we conclude that rule-based approaches are a more flexible alternative as rules can be added without complete knowledge of an underlying formal system. Nevertheless, the rules used in rule-based approaches need to be maintained (revised and grown), resulting in possibly time-consuming and costly work. However, the completeness of such sets of rules can be varied, allowing for an economic trade-off. Therefore, for the purpose of inconsistency management in heterogeneous engineering models, we envision the use of a rule-based approach.

## 14.5 Semantic Web Technologies in a Nutshell

In order to address the aforementioned challenges in an appropriate manner, a highly flexible and maintainable (software) system is required. As a basis to achieve flexibility and maintainability, we envision the application of Semantic Web Technologies. In the following, the core technologies to providing effective and efficient engineering support in the automated production systems domain are presented. For a more detailed overview and introduction to basics of Semantic Web Technologies, please refer to Chap. 3.

### From a Procedural Software System to a Knowledge-Based System

An exclusively procedural software system requires the explicit inclusion of knowledge about the structure and semantics of the various models involved during engineering within the code (Feldmann et al. 2015b). As a consequence, especially when a variety of disparate models is created (cf. challenge 1), maintaining and evolving such a software system is costly: in the worst case, the management of $n$ models requires $n \cdot (n-1)/2$ bi-directional model integrations. A practical realization of a framework for supporting the engineering of automated production systems therefore requires a high degree of flexibility and extensibility. One means to achieve such a flexible and extensible framework is to represent models in a common representational formalism (Estévez and Marcos 2012) and to put appropriate mechanisms for, e.g., identifying and resolving inconsistencies in place. Consequently, we envision a *knowledge-based system* to be used to support the engineering in the automated production systems domain (Feldmann et al. 2015b). Among others, knowledge-based systems typically consist of two essential parts: a *knowledge base* that is used to represent the facts about the world (i.e., the knowledge modeled in the different models) and an *inference mechanism* that provides a set of logical assertions and conditions to process the knowledge base (i.e., to identify and resolve inconsistencies). Further typical parts of knowledge-based systems are the *explanation* and *acquisition* components as well as the *user interface*, which are, however, not in focus of this Chapter.

### Representing Knowledge in Knowledge Bases: RDF(S) and OWL

One formal language used to describe structured information and, hence, to represent knowledge in the context of Semantic Web Technologies is the *Resource Description Framework (RDF)* (World Wide Web Consortium 2014). Originally, the goal of RDF is to allow applications to "exchange data on the Web while preserving their original meaning" (Hitzler et al. 2010), thereby allowing for further processing knowledge. 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 modeling approaches such as class diagrams in that it allows for statements to be made about entities, e.g., *stack is a module and consists of a monostable cylinder and a micro switch*. Such statements about entities are formulated by means of subject–predicate–object triples (e.g., *stack – is a – module*, *stack – consists of – micro switch*), thereby forming a directed graph. An exemplary RDF graph is visualized in Fig. 14.3. To leave no room for ambiguities, RDF makes use of so-called *Unified Resource Identifies* as unique names for entities (e.g., *ex:stack* and *ex:Module*) and properties (e.g., *ex:consistsOf*) being used. By means of so-called RDF vocabularies, collections of identifiers with a clearly defined meaning can be described. 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 *RDF*

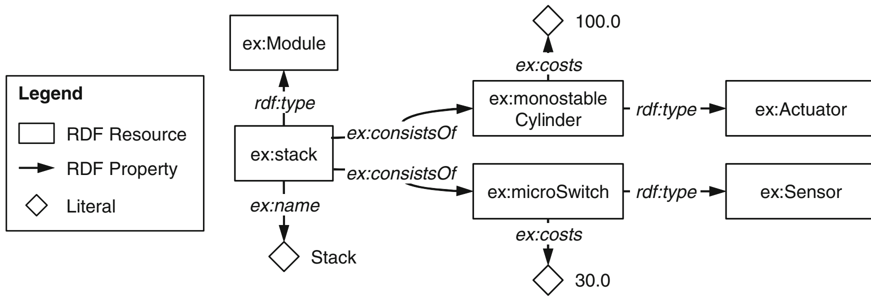



**Fig. 14.3** Exemplary RDF graph

*Schema (RDFS)*. RDF(S) provides language constructs to formulate simple graphs containing class and property hierarchies as well as property restrictions. With its formal semantics, RDF(S) leaves no room for interpretation of what conclusions can be drawn from a given graph, thereby providing standard inference mechanisms for any RDF(S)-compliant graph. RDF(S) 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 (Hitzler et al. 2010). Examples for knowledge that cannot be formulated in RDF(S) are the phrases *Each Module consists of at least one component* and *Components are either actuators or sensors*.

One mean to formulate 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 mean to formulate complex knowledge is the use of the *Web Ontology Language (OWL)* (World Wide Web Consortium 2009), which provides further language constructs defined with description logics based semantics. OWL moreover contains two sub-languages[1] to provide a choice between different degrees of expressivity, scalability, and decidability, namely, OWL Full and OWL DL. Therein, for the purposes of this Chapter, OWL DL[2] enables maximum expressivity while maintaining decidability (Hitzler et al. 2010). The OWL DL formal (description logics based) semantics allow to define what conclusions can be drawn from an OWL DL ontology. For instance, the aforementioned phrases *Each Module consists of at least one component* and *Components are either sensor or actuators* can be formulated as specified in OWL[3] axioms (1) and (2).

MODULE **EquivalentTo** ( *consistsOf* **some** COMPONENT ) (1)

COMPONENT **EquivalentTo** ( SENSOR **or** ACTUATOR ) (2)

[1]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 Chapter.

[2]For reasons of simplicity, we use the term *OWL* when referring to *OWL DL*.

[3]To enhance readability, OWL Manchester Syntax is used throughout the paper.

Therein, the concepts *Module*, *Component*, *Sensor*, and *Actuator* refer to the set of possible individuals that are modules, components, sensors, or actuators. Thereby, typical types of inferences can be drawn from an OWL ontology, e.g.,

- *Satisfiability Checking* identifies, whether a concept is satisfiable. For instance, the question, *Can an instance be both a module and a component?* can be answered by identifying whether the concept MODULE **and** COMPONENT is satisfiable.
- *Subsumption* identifies, whether class hierarchies exist. In our example, the class hierarchy SENSOR **SubClassOf** COMPONENT and ACTUATOR **SubClassOf** COMPONENT can be inferred from axioms (1) and (2).
- *Consistency Checking* identifies whether inconsistencies in the model exist.

### Accessing Knowledge in Knowledge Bases: SPARQL Query Language

A set of specifications providing the means to retrieve and manipulate information represented in RDF(S) (or OWL, respectively) is the *SPARQL Protocol and RDF Query Language* (World Wide Web Consortium 2013). The primary component of the standard is the *SPARQL Query Language*.[4] SPARQL is in many regards similar to the well-known *Structured Query Language (SQL)*, which is supported by most relational database systems.

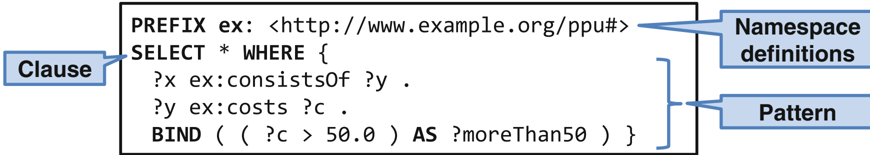
A query consists of three major parts: *namespace* definitions being used within the query, a *clause* identifying the type of the query and a *pattern* to be matched against the RDF data. SPARQL is highly expressive and allows for the formulation of required and optional patterns, negative matches, 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 example for a SPARQL SELECT query is shown in Fig. 14.4. Using this query, the RDF graph in Fig. 14.3 is queried for entities $x$ that consist of an entity $y$, which, in turn, is described by the cost value $c$. By using the BIND form, the Boolean result of the formula $?c > 50.0$ can be assigned to variable *moreThan50*, which denotes, whether the cost value $c$ is greater than *50* or not.

[4]For reasons of simplicity, we use the term *SPARQL* when referring to *SPARQL Query Language*.

**Exemplary query**




**Results of query execution**

| ?x | ?y | ?c | ?moreThan50 |
|---|---|---|---|
| ex:stack | ex:monostableCylinder | 100.0 | true |
| ex:stack | ex:microSwitch | 30.0 | false |

**Fig. 14.4** Exemplary SPARQL query (*top*) and results when executing the query

## 14.6 Use Cases for Applying Semantic Web Technologies in the Automated Production Systems Domain

Within the following, three distinct use cases are presented, which aim at supporting the engineering in the automated production systems domain by means of Semantic Web Technologies.

### *14.6.1 Use Case 1: Ensuring the Compatibility Between Mechatronic Modules*

Automated production systems are characterized by a multitude of mechanical, electrical/electronic and software components with tight interrelations between them. In order to facilitate the development of automated production systems and reduce costs of the engineering process, companies usually define mechatronic modules including components from different disciplines, which can be reused in various systems.

During the life cycle of such a system, frequent changes have to be carried out to different system components or modules, e.g., if new customer requirements have to be fulfilled or if specific components/modules have to be replaced but are not available on the market any more. A challenge in carrying out changes during the life cycle of an automated production system is to ensure the compatibility of the exchanged component/module with the existing system (e.g., regarding data ranges of specified properties, type compatibility, etc., (Feldmann et al. 2014a)). Lacking consideration of change influences can lead to further necessary changes in the system, which are costly and can prolong the downtime unnecessarily.

Therefore, this use case describes how a model-based approach can be used to analyze changes before they are implemented in the real system (Feldmann et al. 2014a). Consequently, we aim at combining such a model-based approach with

Semantic Web Technologies to provide the means (1) to identify compatible modules in case a module needs to be replaced, and (2) to identify and resolve incompatibilities in a given system configuration. A more detailed description of this use case can be retrieved from (Feldmann et al. 2014a).

### Overall Concept

In order to specify the relevant aspects for checking the system for incompatibilities, an information model is defined that contains the information necessary for identifying whether two modules are compatible or not. By that, any Model-Based Engineering (MBE) approach can be combined with our compatibility information model, which can directly be used to compute the formal knowledge base. Figure 14.5 shows the relation between the MBE approach, the formal knowledge base and the information model with its elements and relations. As shown in the information model, structural aspects of mechanical, electrical/electronic, and software components, which can be combined to mechatronic modules, and the respective interfaces in the different disciplines are considered for the compatibility check. Furthermore, the functionalities, which are fulfilled by a component/module, are considered.

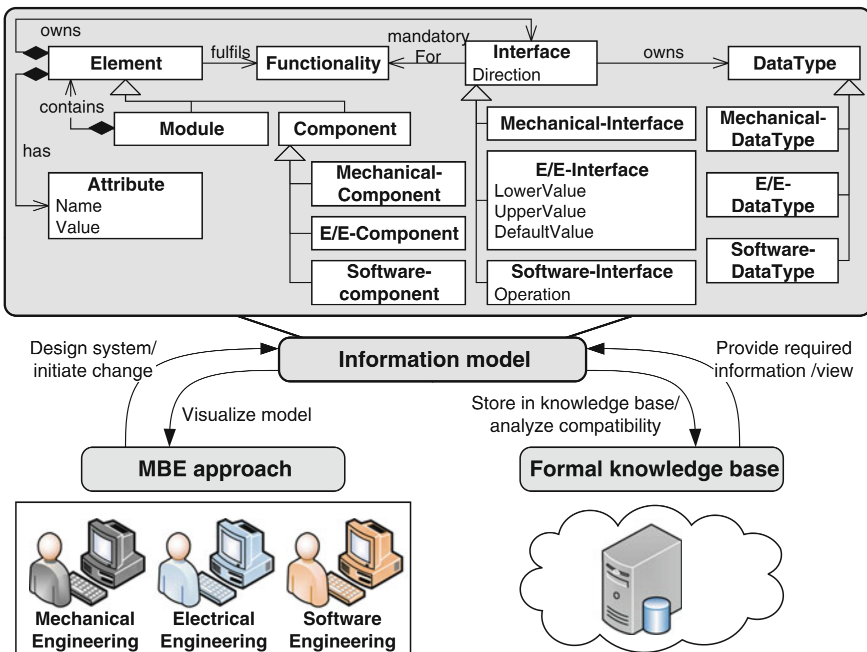



**Fig. 14.5** Combination of MBE approach and a formal knowledge base to analyze changes, extended from Feldmann et al. (2014a)

### Visual Model for Modeling Systems Comprehensibly

Regarding the visual model, Model-Based Engineering (MBE) approaches gained more and more influence over the past years. Especially for systems engineering, the *Systems Modeling Language (SysML)* (Object Management Group 2012) was developed as a graphical modeling language to represent structural and behavioral aspects during development. Through specific modeling approaches, based on SysML, the relevant aspects for analyzing changes can be integrated into the model. In this use case, SysML4*Mechatronics* (Kernschmidt and Vogel-Heuser 2013) is used as modeling language, as it was developed specifically for the application of mechatronic production systems. An exemplary SysML4*Mechatronics* model for the PPU application example is shown in Fig. 14.6. Next to the mechatronic modules (*stack*, *crane*, *stamp*, and *ramp*), the *bus coupler*, the *PLC* and the *mounting plate* are depicted as separate blocks, which are required by the modules, e.g., all modules are connected to the mounting plate of the system mechanically and, if required, communicate through a Profibus DP interface.

### OWL and SPARQL for Compatibility Checking

For identifying incompatibilities between mechatronic modules, we argue that two disparate types of compatibility rules exists

- *Inherent compatibility rules* apply for arbitrary types of automated production systems and must not be violated by any system under study. Examples for such inherent compatibility rules are type and direction compatibility.
- *Application-specific compatibility rules* apply within a given context (e.g., for specific types of systems or for a concrete project). For instance, project-specific naming conventions are often applied for specific projects or applications.

In order to allow for flexibly maintaining and extending a software system for checking compatibility of mechatronic modules, we envision the application of

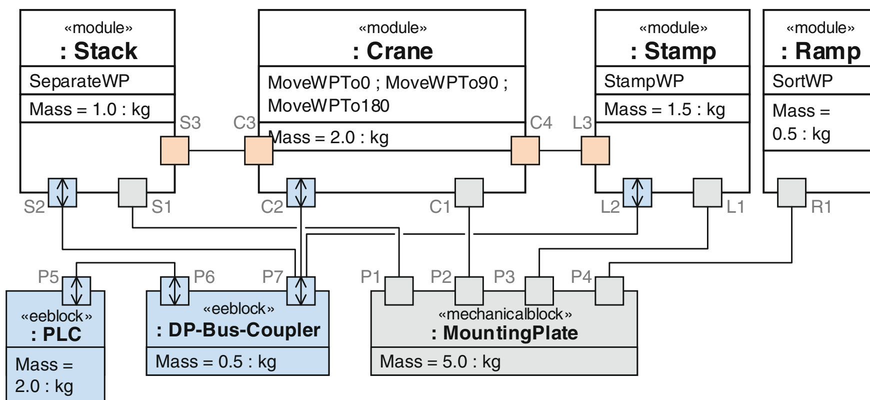
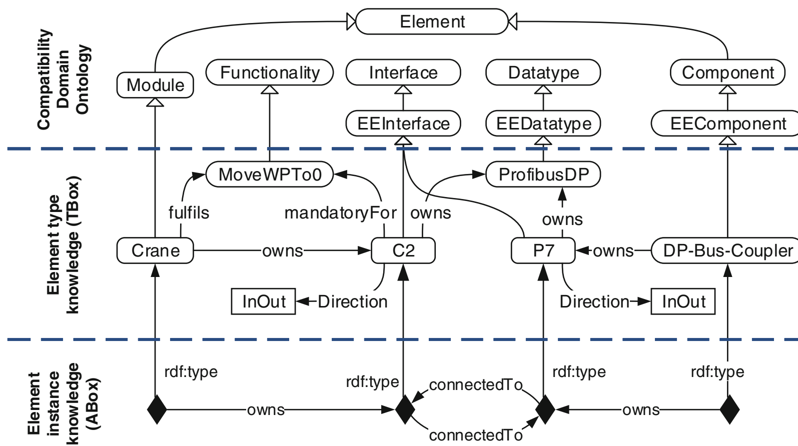



**Fig. 14.6** Exemplary SysML4*Mechatronics*model for the PPU application example

OWL, which provides the means to formulate the knowledge in a compatibility information model shown in Fig. 14.5, and SPARQL, which allows identifying whether certain compatibility rules are violated or not. Within OWL, a domain ontology is used, which defined the concepts and relations necessary to represent the knowledge in our compatibility information model. Using this domain ontology, knowledge on available modules, interface types, etc., can be formulated in the ontology's terminological knowledge. Accordingly, knowledge on the instances available for the system under study is represented in the ontology's assertional knowledge. As a consequence, SPARQL queries can easily be formulated using the terms defined in the compatibility domain ontology. We argue that SPARQL queries can, hence, be defined, maintained, and managed more efficiently using such a domain ontology.

The representation of the crane application example in an OWL ontology as well as some exemplary compatibility rules are illustrated in Fig. 14.7. Using SPARQL SELECT queries, incompatibility patterns are described; any result returned by querying the ontology stands for incompatible elements within the model. Through




| Inherent compatibility rules | | Application-specific compatibility rules | |
|---|---|---|---|
| 1 – Interfaces' data types | **SELECT** *?x ?y* **WHERE** {<br>*?x* :connectedTo *?y* .<br>*?x* **a** ( :Interface **and** :owns **some** *?xType* ) .<br>*?y* **a** ( :Interface **and** :owns **some** *?yType* ) .<br>**FILTER** ( *?xType* != *?yType* ) . } | 3 – Naming conventions | **SELECT** *?x* **WHERE** {<br>*?x* **a** :Element ; :Name ?n .<br>**FILTER** ( ! regex ( ?n , „^[^0-9].*“) ) . } |
| 2 – Interfaces' data ranges | **SELECT** *?x ?y* **WHERE** {<br>*?x* :connectedTo *?y* . *?y* :UpperValue *?yUpp* .<br>*?x* :UpperValue *?xUpp* ; :Direction *?xDir* .<br>**FILTER** ( *?xDir* = „In“‖*?xDir* = „InOut“ ) .<br>**FILTER** ( *?yUpp* > *?xUpp* ) . } | | |

**Fig. 14.7** Representation of the crane example in OWL (*top*) and exemplary compatibility rules formulated in SPARQL (*bottom*)

queries 1 and 2, it can be identified whether data types and ranges of two ports are compatible or not. Query 3 defines an application-specific compatibility rule that requires entities' names not to start with a number.

### Proof-of-Concept Implementation

The conceptual architecture introduced previously was realized in terms of a proof-of-concept implementation, see Fig. 14.8. The models are defined in the Eclipse Modeling Framework (EMF).[5] The necessary model transformations, i.e., between SysML4*Mechatronics*, the compatibility information language and OWL, were realized by means of the Query/View/Transformation Operational (QVTo) standard (Object Management Group 2011) and executed using the Eclipse QVTo implementation.[6] For OWL, a meta model was developed in the well-known Ecore format based on the respective W3C standard (World Wide Web Consortium 2009). Subsequent XSL Transformations allow for transforming between the EMF-specific XML format and the respective tool-specific XML formats. Pellet[7] was used as the reasoner and querying engine for executing the queries.

### Validation

In the current state (shown in Fig. 14.6), the crane has four distinct interfaces for the connection to the rest of the system. During the life cycle of the system, the crane has to be replaced. Due to the aspired shift to a Profinet bus system, a crane module with Profinet interface shall be used for the exchange. Except for the change of the bus system, the crane module shall fulfill the same functionalities. Figure 14.9 shows the SysML4*Mechatronics* model of the system with the replaced crane module.

Having transformed the modeled information into the OWL DL ontology of the formal knowledge base, the respective SPARQL queries can be executed. Using the

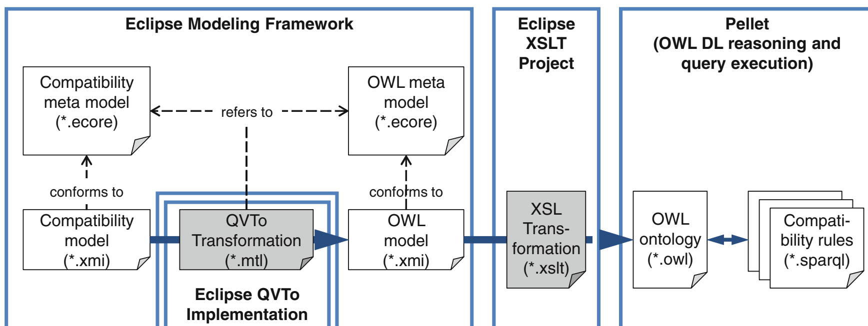



**Fig. 14.8** Overview on the architecture for compatibility checking

[5] https://eclipse.org/modeling/emf/, retrieved on 12/11/2015.

[6] https://wiki.eclipse.org/QVTo, retrieved on 12/11/2015.

[7] https://github.com/complexible/pellet, retrieved on 12/11/2015.

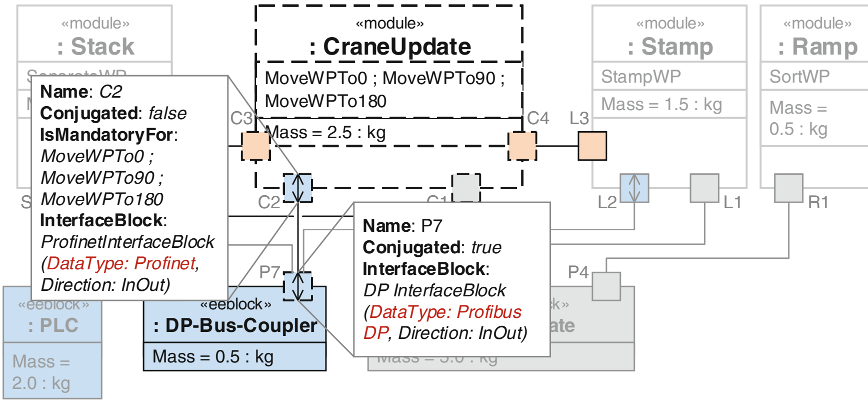



**Fig. 14.9** Exchanged crane module (represented as SysML4*Mechatronics* module)

example queries as described above, an incompatibility between the Profinet and the Profibus DP interface can be identified (cf. query 1), as the respective datatypes are not compatible. Queries 2 and 3 do not identify incompatibilities, as data ranges are not applicable for the illustrated example (cf. query 2) and no violations of the naming conventions for entities can be identified (cf. query 3).

### *14.6.2 Use Case 2: Keeping Requirements and Test Cases Consistent*

In the automated production systems domain, requirements are often specified in an informal and textual manner (Runde and Fay 2011). This may lead to ambiguous and erroneous interpretations of requirements. As a consequence, inconsistencies in these requirements specifications may arise. These inconsistencies are often identified late and can, thus, lead to additional costs (Heitmeyer et al. 1996). In addition, the automatic generation of test cases is not yet state of the art in the automated production systems domain (Hametner et al. 2011): test cases are often defined during design and not during requirements specification. To address these challenges, this use case introduces an integrated approach for systematic requirements and test case design as well as for ensuring consistency between requirements and test cases (Feldmann et al. 2014b). On the one hand, the approach makes use of a modeling approach for specifying requirements and test cases in a semi-structured manner. On the other hand, Semantic Web Technologies are applied to allow for consistency checking of the models and, thus, of the modeled requirements and test cases. For a detailed description of the models being used in this use case, please refer to (Feldmann et al. 2014b).

## Modeling Requirements and Test Cases

The approach for modeling requirements and test cases systematically and graphically consists of three primary modeling elements, namely, *feature*, *requirement*, and *test case* (cf. Fig. 14.10).

**Features**. Features refer to a plant component or functionality, e.g., the capability of transporting or detecting a work piece. In the PPU application example, the feature *PPU* as well as its sub-features *Stack*, *Stamp*, *Ramp*, and *Crane* are defined. In addition, features are characterized by parameters that either represent sensors or actuators of the system under study (namely, in- and out-parameters) or parameters that describe the product and environment. By means of these parameters, requirements on features can be further specified. For instance, the crane feature is further defined by the in-parameters *source*, which defines the source of the transport function (either "ramp", "stack," or "stamp"), *velocity*, which characterizes the crane module's velocity, as well as the work piece's *Mass* and *type*. Out-parameters

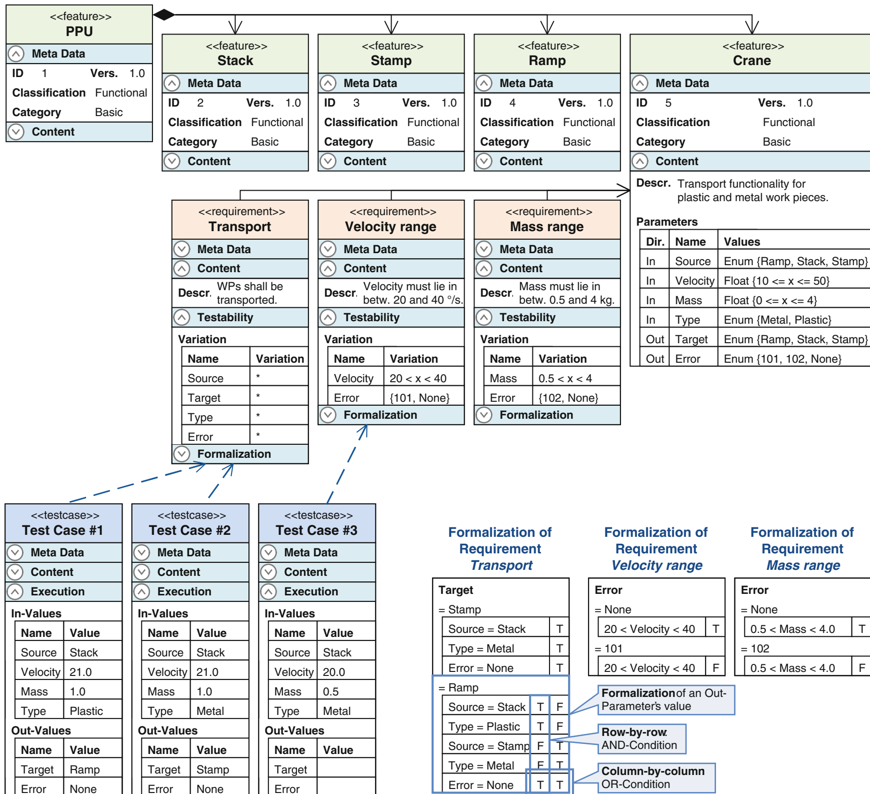



**Fig. 14.10** Example requirements and test case model for the PPU application example, extended from Feldmann et al. (2014b)

that are available for the crane feature are *target*, which specifies the target of the transport function, and *error*, which defines an error code (either "101", "102," or "None").

**Requirements**. A requirement is associated to a set of features, thereby characterizing their intended behavior by means of parameter restrictions. In our PPU application example, three distinct requirements are defined: a *transport* requirement defines how work pieces are transported, and the *velocity* and *mass range* requirements specify applicable ranges for the crane's velocity and a work piece's mass. Parameter variations in requirement specifications define in which range a parameter needs to be tested for a given requirement. For instance, the crane must be able to operate for work pieces that weigh between *0.5* and *4* kg. These variations can furthermore be used for generating test cases, e.g., by generating all possible combinations of in-parameter values with respect to their variations. As requirements can either be functional or nonfunctional, the modeling approach allows modelers to either textually specify a (nonfunctional) requirement or to specify their (functional) requirements in a semi-structured manner by means of formalizations. These formalizations provide the means to define for which *condition* a certain parameter holds a specific value, i.e., which *implication* can be drawn from a given condition. The implication is thereby equivalent to setting an out-parameter's value; the condition is a set of logical phrases that can be evaluated to a truth value.

For the graphical representation of the formalizations, a subset of SPECTR-ML (Leveson et al. 1999) is used and represented by means of truth tables; an example for formulating the functional requirements of the crane feature is illustrated in Fig. 14.10. Each truth table represents an out-parameter and contains the parameter's possible values, i.e., possible implications that can be drawn from given conditions. For instance, the *target* parameter can be set to either one of the values "stamp" or "ramp". For each implication, a set of conditions can be defined by means of rows in the truth tables. In each row of the truth tables, the far left column refers to the logical phrase. Each of the other columns refers to a conjunction of logical phrases and contain the truth values related to the logical phrase, i.e., true (T), false (F), and don't care (*). Consequently, as can be seen in Fig. 14.10, a column evaluates to true if all of its related rows match the truth values of the associated predicates. For instance, regarding the *target* parameter, the following formalizations are specified in Fig. 14.10:

- The *target* parameter must be set to *stamp* if, given that no error occurs, a *metal* work piece is available at the *stack*.
- The *target* parameter must be set to *ramp* if, given that no error occurs, either a *plastic* work piece is available at the *stack* or a *metal* work piece is available at the *stamp*.

**Test Cases**. To ensure that a certain requirement is fulfilled, test cases are defined for a given set of requirements. These test cases can either be specified manually (e.g., for non-functional requirements) or generated automatically (e.g., from formaliza-

tions of functional requirements). A test case therefore consists of a set of parameter values to be considered for a test: based on given in-parameter values, expected out-parameter values are defined. Three exemplary test cases for the PPU application example are shown in Fig. 14.10.

**Ensuring Consistency Between Requirements and Test Cases**

Although the previously introduced modeling concept provides a starting point for discussing and communicating the system under study, potential inconsistencies in the requirement and test case specification may arise. Among others, the following types of inconsistencies are likely to occur:

- *Inconsistencies between features and (functional) requirements*: Parameters are specified by valid parameter ranges in the feature. These parameters are used and further detailed in requirements associated to the feature by means of parameter variations that define in which range a parameter must be tested. In order for a model to be valid, the parameter ranges defined in the feature must be consistent with the parameter variations specified for a requirement.
- *Inconsistencies in between (functional) requirements*: The set of requirements defined for a feature is formalized by means of logical conditions and implications. Consequently, the entire set of requirements defined for a feature describes the system behavior as desired by the modeler. These requirements may consist of contradicting formalizations, leading to potential inconsistencies between requirements. Consistency must therefore be ensured between the formalizations of a feature's requirements.
- *Inconsistencies between (functional) requirements and test cases*: Based on the parameter variations defined in a feature's requirements, test cases can be generated, e.g., by identifying all permutations of available in-parameters. Nevertheless, for each given permutation of in-parameters, a unique combination of out-parameters must be determinable in order for a model to be consistent.

It is obvious that all of these types of inconsistencies can be identified as a logical consequence of contradiction statements. Hence, the use of OWL, its description logics based semantics and the inference types that can be drawn from an OWL model (e.g., whether a concept is satisfiable or whether inconsistencies in the model exist) is an appropriate solution approach. As a consequence, based on the parameter ranges in a feature we formulate a set of possible feature states by means of an OWL concept. Each implication defined in the requirements' truth tables can then be specified as a sub-concept of this feature state—an OWL reasoner can consequently determine whether or not this concept is satisfiable, i.e., whether the requirement is consistent to the feature. If implications are over-specified (see, for instance, the specification of the implication *Error = None* in Fig. 14.10), the respective implication is defined as the intersection of the existing set of implications. If the set of implications is inconsistent, an OWL reasoner identifies the model to be inconsistent, i.e., that there is an inconsistency between the requirements. Finally, test cases are regarded as states of the feature concept and, hence, if contradictions between

a test case and the requirement occur, an OWL reasoner identifies the model to be inconsistent. Consequently, using such an OWL model, all aforementioned types of inconsistencies can be identified.

In the resulting OWL model, parameters defined in respective features are represented by OWL data properties. As we assume that for each parameter only one value exists in a feature's state, the corresponding data properties are defined to be functional. Consequently, we formulate a feature's state as an OWL concept, which is equivalent to the intersection of the respective parameter ranges defined in the feature. For instance, the crane feature's state can be formulated within a respective concept CraneState (cf. axiom (3)).

CraneState **EquivalentTo** ( *Error* **some** {"101", "102", "None"} )
**and** ( *Source* **some** {"Stack", "Stamp", "Ramp"} )
**and** ( *Target* **some** {"Stack", "Stamp", "Ramp"} )
**and** ( *Type* **some** {"Metal", "Plastic"} )
**and** ( *Velocity* **some float**[≥10, ≤50] ) **and** ( *Mass* **some float**[≥0, ≤4] ) (3)

Respectively, formalizations in truth tables of requirements are defined as OWL concepts being sub-concepts of the feature's state. For instance, the formalization of requirements *velocity* and *mass range* are specified in axioms (4)–(6):

Error101 **EquivalentTo** CraneState **and** ( **not** *Velocity* **some float**[>20, <40] ) (4)

Error102 **EquivalentTo** CraneState **and** ( **not** *Mass* **some float**[>0.5, <4] ) (5)

ErrorNone **EquivalentTo** CraneState **and** ( *Velocity* **some float**[>20, <40] )
**and** ( *Mass* **some float**[>0.5, <4] ) (6)

Accordingly, the implications being defined in the requirements' truth tables can be formulated as the necessary conditions for the respective concepts, see axioms (7)–(9).

Error101 **SubClassOf** ( *Error* **value** "101" ) (7)

Error102 **SubClassOf** ( *Error* **value** "102" ) (8)

ErrorNone **SubClassOf** ( *Error* **value** "None" ) (9)

The formalizations of the requirement *transport* are specified accordingly. Using the model, it can be identified whether contradictions exist between requirements and features as well as in between requirements. In the present PPU application example, it is obvious that no inconsistency exists and, hence, an OWL reasoner infers the model to be consistent. However, if we formulate the test cases as depicted in Fig. 14.10 to be instances of the feature state's concept CraneState, an inconsistency can be detected for test case 3: as both the *velocity* and *mass* parameters violate the formalizations imposed in the *velocity* and *mass range* requirements, an OWL reasoner infers the *error* parameter to hold both the values "101" and "102". Nevertheless, as all properties were defined to be functional, the OWL reasoner identifies an inconsistency. This ambiguity is identified and the engineer is notified that the requirements need to be specified further; it is identified that the test case 3 cannot be consistent to the requirements.

**Proof-of-Concept Implementation**

In order to evaluate the concept for modeling requirements and test cases and for ensuring consistency, a proof-of-concept implementation was developed. An overview on the architecture of the prototypical tool is given in Fig. 14.11. Model instances are created by the modeler using the Eclipse Modeling Framework (EMF). Therefore, a metamodel was defined in the well-known Ecore format, which allows for providing a simple tree editor in EMF. The mapping to the respective OWL ontology was defined by means of the MOF Model to Text Standard (MOFM2T) (Object Management Group 2008), which can, e.g., be executed in the Acceleo[8] MOFM2T implementation. It has to be noted that the domain ontology (that, e.g., defines the base properties and classes to be used) is independent from the created models and, hence, is imported by the respective model-specific application ontologies. The resulting OWL ontology can then be processed by available OWL reasoners—in our implementation, Pellet was used.

## *14.6.3 Use Case 3: Identifying Inconsistencies in and Among Heterogeneous Engineering Models*

During the development and operation of automated production systems, a multitude of stakeholders from different disciplines is involved. In order for the specific concerns of these stakeholders to be addressed, views on the system under study are

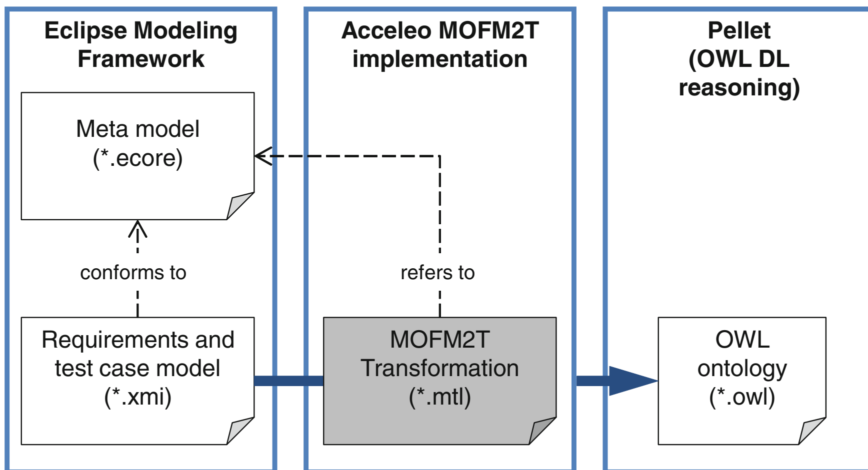



**Fig. 14.11** Overview on the architecture for consistency checking among features, requirements and test cases

[8] http://www.eclipse.org/acceleo/, retrieved on 12/11/2015.

formed. To adequately address these concerns, a variety of different formalisms, modeling languages and tools is necessary (Broy et al. 2010). Two distinct formalisms that address different stakeholders' concerns were shown in the previous sections: one for the detailed design of automated production systems (cf. use case 1) and one for requirements and test case management (cf. use case 2). Nevertheless, although it is considered good practice to separate concerns as much as possible, completely separating concerns is impossible. As a consequence, some concerns are addressed by the various stakeholders and, hence, lead to overlaps in the models. An example of such a model overlap can be seen in use cases 1 and 2: both models refer to the different modules of the PPU, therefore specifying overlapping information on the system under study and, hence, leading to the potential for inconsistencies to occur. Consequently, it is inevitable to ensure that the set of models is free of inconsistencies, which is being addressed in this use case. A detailed description of the use case can be found in (Feldmann et al. 2015b).

**Using RDF to Represent Heterogeneous Models in a Common Formalism**

As argued beforehand, effectively and efficiently handling inconsistencies necessitates a high degree of flexibility. Consequently, we argue that representing the heterogeneous models in a common representational formalism is inevitable for any inconsistency management framework to be economical.

Information in knowledge bases is typically represented using predicated statements about entities (Giarratano and Riley 1994). This is similar to how information and knowledge are represented in the abstract syntax of models (Herzig et al. 2011): For instance, one fact encoded in use case 1 asserts that the PPU is composed of (among other modules) a crane. This fact can be represented as *PPU composedOf crane*, where *PPU* and *crane* are predicable entities, and *composedOf* a predicate. Consequently, we argue that Semantic Web Technologies are an appropriate way to represent the knowledge modeled in disparate modeling formalisms. In the following, we use RDF(S) as the common representational formalism; Fig. 14.12 shows an excerpt of the RDF(S) representations of the models being used in use cases 1 and 2.

In a similar manner, overlaps can be captured in RDF. Given that a common representational formalism is used, one can formulate statements in which entities from any model are referenced. This can be seen as an additional model, describing relationships among the different models. For instance, given an RDF namespace *overlaps* for this additional model, and given the definition of an RDF property *equivalentTo*, which is to be used for defining the synonymy for two predicable entities, the statement *sysml4mech:expVelocity overlaps:equivalentTo req:reqVelocity* (see Fig. 14.12) expresses the fact that both entities are semantically equivalent. Such semantic relations can either be defined manually and a priori or by applying appropriate inference mechanisms.

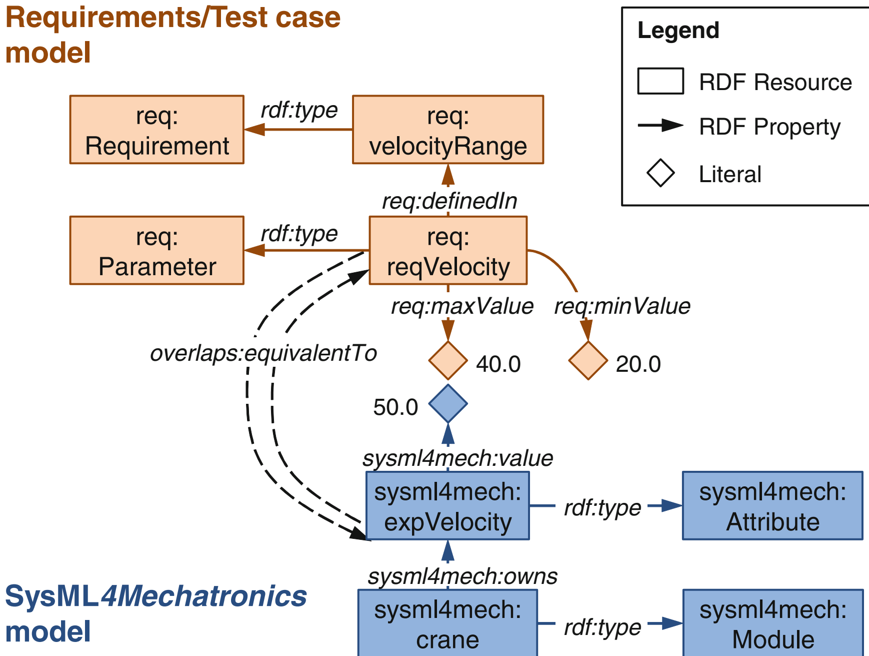



**Fig. 14.12** Excerpt of the RDF representation for the use case models

### Vocabularies for Identifying Cross-Model Inconsistencies

Given that RDF(S) as the common representational formalism is used, inconsistencies among the heterogeneous models can be identified. However, with an increasing number of models involved, the number of necessary integration between models arises, making mechanisms for efficient and effective handling of inconsistencies necessary. Therefore, we argue that a common terminology is required that bridges the gap between different domain models and, thus, (1) provides a common syntax, and (2) allows for defining the semantic relations between the modeled information. We hypothesize that—at least at some level of (semantic) abstraction—there exist concepts common to specific domains, which can be represented using different languages, and that concepts exist that are common to all domains, e.g., the term *value* that is used in both use cases 1 and 2. Using such a *base vocabulary* (cf. Fig. 14.13), some common inconsistencies among and between multiple domain models can be managed. Clearly, the definition of *semantic overlaps* may not always require the full expressiveness of domain models; thus, the base vocabulary is semantically much weaker than *domain vocabularies* or *language vocabularies* and represents a "common denominator" to all other vocabularies. The base vocabulary ideally remains unchanged, while integrating a further type of domain model necessitates only a respective novel RDF representation and the definition of a *semantic mediation* between the novel domain vocabulary and the base vocabulary.

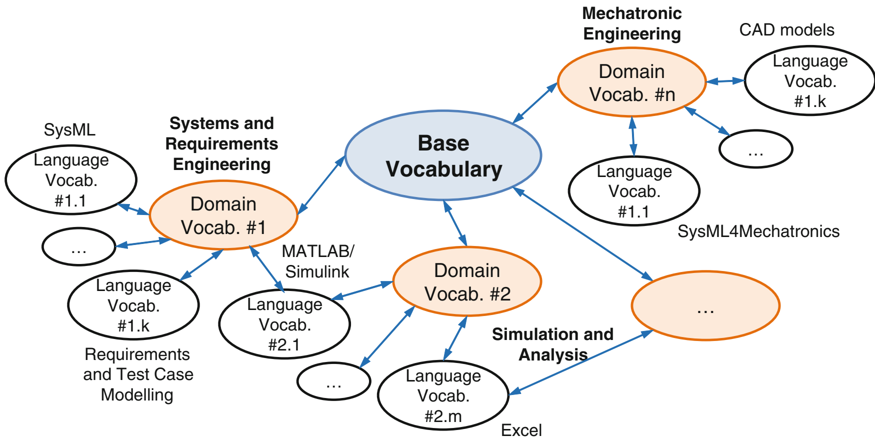



**Fig. 14.13** Semantic mediation between language, domain, and base vocabularies

The result of a mediation from the respective use case models to the base vocabulary is illustrated in Fig. 14.14. As can be seen, attributes' values are mediated to the common concept *Constraint*, which, in turn, can be either an *EqualsConstraint* (attribute's value is specified to have a specific value), a *LessThanConstraint* (attribute's value is specified to be less than a specific value) and a *GreaterThanConstraint* (attribute's value is specified to be greater than a specific value). By that, inconsistency rules can be formulated by referring to the concepts defined in the base vocabulary.

## SPARQL Queries for Identifying Inconsistencies

For engineering of physical systems such as automated production systems, it is impossible to say whether or not such systems are fully consistent (Herzig et al. 2011). The main reason is the lack of perfect knowledge about the processes and the

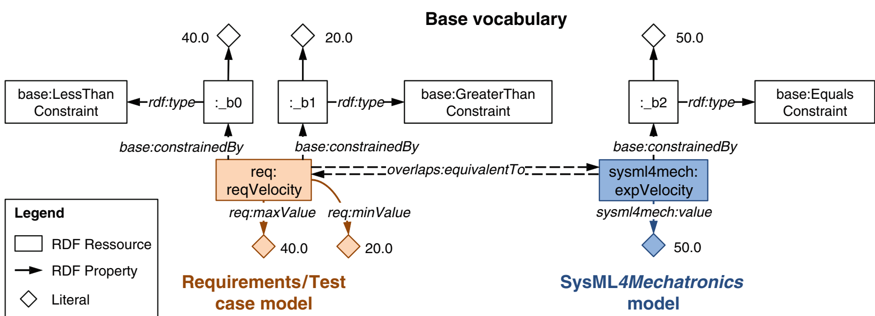



**Fig. 14.14** Result of mediating the use case models to the base vocabulary

phenomena in nature (e.g., regarding precision of manufacturing processes). The best one can do is to identify specific types of inconsistencies defined a priori by an expert. Such inconsistencies can, for instance, result from logical contradictions (e.g., over- or under-specifications of attributes' values) or from violations of agreed heuristics and guidelines (Feldmann et al. 2015b). Thinking of types of inconsistencies in this manner makes the representation of inconsistencies as rules, i.e., in the form *IF condition THEN action*, feasible. Consequently, to identify specific instances of inconsistencies according to the known types of inconsistencies, we use SPARQL SELECT queries to (1) formulate the context of, and conditions for inconsistencies using a graph pattern (the *condition*) and (2) retrieve a list of those elements which were checked for a type of inconsistency and either met, or did not meet the condition for inconsistency (the *action*). In this manner, inconsistency checks are similar to unit tests.

Two exemplary inconsistency rules that serve the purpose of identifying, whether the constraints imposed in Fig. 14.14 contradict each other or not, are illustrated in Fig. 14.15. Therein, query 1 matches any two entities $x$ and $y$ identified as semantically equivalent, that are constraint by an *EqualsConstraint* (entity $x$) and a *LowerThanConstraint* (entity $y$), respectively. The comparison of the entities' values, namely, $?xVal < ?yVal$, is bound to the variable *isInconsistent* and denotes whether the constraints are inconsistent or not. Query 2 is formulated accordingly to allow for comparing *EqualsConstraints* with *GreaterThanConstraints*.

In the presented use cases 1 and 2, the value of the SysML4*Mechatronics* property *expVelocity* is defined to be 50.0, whereas the value of *reqVelocity* in the requirements and test case model is defined to be greater than or equal to 20.0 as well as lower than or equal to 40.0. Consequently, using the queries 1 and 2, it can be identified that the respective *LowerThanConstraint* is violated (cf. query 1), whereas the *GreaterThanConstraint* is fulfilled.

**Query #1**

```
PREFIX : <http://www.example.org/base/ns#>
PREFIX overlaps: <http://www.example.org/overlaps/ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?x overlaps:equivalentTo ?y .
  ?x :constrainedBy [ rdf:type :EqualsConstraint ; :value ?xVal ] .
  ?y :constrainedBy [ rdf:type :LowerThanConstraint; :value ?yVal ] .
  BIND ( ( ?xVal > ?yVal) AS ?isInconsistent ) }
```

**Query #2**

```
PREFIX : <http://www.example.org/base/ns#>
PREFIX overlaps: <http://www.example.org/overlaps/ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?x overlaps:equivalentTo ?y .
  ?x :constrainedBy [ rdf:type :EqualsConstraint ; :value ?xVal ] .
  ?y :constrainedBy [ rdf:type :GreaterThanConstraint; :value ?yVal ] .
  BIND ( ( ?xVal < ?yVal) AS ?isInconsistent ) }
```

**Fig. 14.15** Exemplary inconsistency queries for the use case models

**Proof-of-Concept Implementation**

As a proof-of-concept, a technology demonstrator was developed to demonstrate and evaluate the technical feasibility and viability of the conceptual approach. An overview on the basic architecture of the demonstrator is illustrated in Fig. 14.16. For the knowledge base, the RDF triple store Apache Fuseki[9] was used. For the purpose of demonstrating the technology, we assume that a mechanism exists for automatically transforming the models into an RDF representation (cf. use cases 1 and 2). The mediation between the different vocabularies was formulated and performed using the Apache Jena Rule Reasoning Framework.[10] The mechanism for identifying inconsistencies is composed of a set of SPARQL queries, as well as a SPARQL-compliant query engine. For the latter we have used ARQ,[11] which is also a part of the Apache Jena framework. Within the control and representation layer of our technology demonstrator, the queries can be managed, handed to the query engine and the query results can be interpreted and visualized to the user.

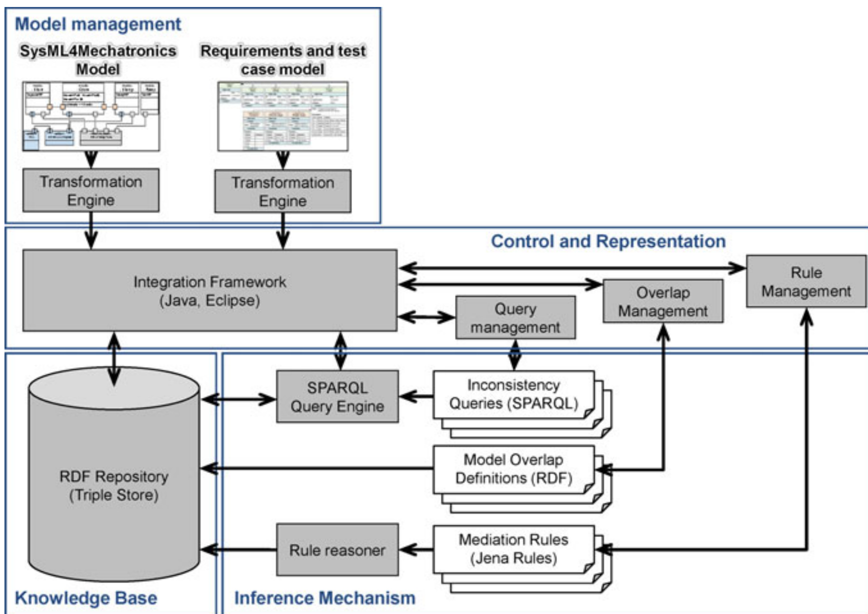



**Fig. 14.16** Technology demonstrator for managing inconsistencies in heterogeneous models, extended from Feldmann et al. (2015b)

[9]http://jena.apache.org/documentation/serving_data/, retrieved on 12/11/2015.
[10]http://jena.apache.org/documentation/inference/, retrieved on 12/11/2015.
[11]http://jena.apache.org/documentation/query/, retrieved on 12/11/2015.

## 14.7 Conclusion and Directions for Future Research

This Chapter is motivated by the challenge of developing modern automated production systems that meet the requirements to manufacture smaller lot sizes up to customer specific products. Such systems have to enable changes in all phases of the system life cycle, to react on new customer or system requirements. Consequently, it is inevitable to support the engineering of such automated production systems by means of, e.g., ensuring consistency. As a basis to overcome this challenge, the Chapter presented three distinct use cases that make use of Semantic Web Technologies as one means to apply logical inference and, by that, to identify inconsistencies in and between the models. In particular, compatibility checking between mechatronic modules, consistency checking between requirements and test cases, as well as, inconsistency management between heterogeneous models were introduced in this Chapter. Consequently, we argue that Semantic Web Technologies can be used to support the engineering of automated production systems. Especially the application of a knowledge-based system, in which Model-Based Engineering is combined with Semantic Web Technologies, provides an integrated approach in which engineers can apply the notations and modeling approaches that are common to them.

Although some of the challenges introduced at the beginning of this Chapter can be addressed by the presented concepts, clearly much research is left to be done.

**Research Direction 1**: **Identification of semantic overlaps.** In our use cases, we assumed that semantic overlaps between model entities were defined a priori by a human. However, for more complex models and systems, we expect that manually defining and managing overlaps is too costly. However, explicit knowledge of semantic overlaps is indispensable for finding most types of inconsistencies. One means to identify semantic overlaps was proposed in (Herzig and Paredis 2014) using a probabilistic reasoning approach. We moreover argue that background knowledge of the domain (e.g., of the automated production systems domain) can be used to efficiently identifying and defining semantic overlaps.

**Research Direction 2**: **Flexible definition and execution of inconsistency rules.** As shown in the use cases, RDF(S), OWL and SPARQL provide the means to formulate and check for different types of inconsistencies. However, we expect that further types of inconsistencies exist, making the investigation of the appropriateness of the mechanism for identifying inconsistencies necessary. Furthermore, we argue that dependencies between different types of inconsistencies occur. For instance, in some cases, the detection of one inconsistency can make the execution of a second inconsistency rule obsolete. Hence, preconditions for executing inconsistency rules needs to be incorporated in future research.

**Research Direction 3**: **Support in resolving inconsistencies.** The ultimate goal of inconsistency management is to support stakeholders in resolving inconsistencies. Consequently, besides the visualization of detected inconsistencies, support for tracing and deciding on how an inconsistency should be resolved is indispensable. In

this way, the development of automated production systems is facilitated through the use of Semantic Web Technologies and thus, enables engineers to create less failure-prone systems.

**Research Direction 4**: **Estimating the suitability of available techniques for inconsistency management.** In this Chapter, Semantic Web Technologies were used as the core technology for inconsistency management in the automated production systems domain. Further approaches that make use of Semantic Web Technologies are, e.g., the works of Kovalenko et al. (2014) that make use of OWL and SPARQL as well as Steyskal and Wimmer (cf. Chap. 13) that make use of the Shapes Constraint Language (SHACL) (World Wide Web Consortium 2015), which is a current working draft of the W3C. Further technologies that are being used for inconsistency management are, e.g., the Object Constraint Language (OCL) (Object Management Group 2014) as well as XML-based methodologies. Surely, each of these technologies has its advantages as well as disadvantages regarding, e.g., usability, comprehensibility as well as scalability and performance for different types and sizes of models. One aspect to be addressed in future work is, hence, to identify, which technology is most suitable for which use case.

**Acknowledgments** This work was supported in part by the German Research Foundation (DFG) Collaborative Research Centre 'Sonderforschungsbereich SFB 768—Managing cycles in innovation processes—Integrated development of product-service systems based on technical products'. Moreover, parts of this work were developed as part of the IGF-project 17259 N/1 of the Deutsche Forschungsgesellschaft für Automatisierung und Mikroelektronik (DFAM) e.V., funded by the AiF as part of the program to support cooperative industrial research (IGF) with funds from the Federal Ministry of Economics and Technology (BMWi) following an Order by the German Federal Parliament. We moreover thank Christiaan J.J. Paredis, Sebastian J.I. Herzig and Ahsan Qamar (Georgia Institute of Technology) for their support and fruitful discussions.

## References

Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D.: Seamless model-based development: from isolated tools to integrated model engineering environments. Proc. IEEE **98**(4), 526–545 (2010). doi:10.1109/JPROC.2009.2037771

Estévez, E., Marcos, M.: Model-based validation of industrial control systems. IEEE Trans. Ind. Inf. **8**(2), 302–310 (2012). doi:10.1109/TII.2011.2174248

Feldmann, S., Kernschmidt, K., 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 (2014a). doi:10.1016/j.procir.2014.01.140

Feldmann, S., Rösch, S., Legat, C., Vogel-Heuser, B.: Keeping requirements and test cases consistent: towards an ontology-based approach. In: IEEE International Conference on Industrial Informatics (2014b). doi:10.1109/INDIN.2014.6945603

Feldmann, S., Herzig, S.J.I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J.J., Vogel-Heuser, B.: A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study. In: IEEE International Conference on Automation Science and Engineering (2015a). 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., Vogel-Heuser, B.: Towards effective management of inconsistencies in model-based engineering of automated production systems. In: IFAC Symposium on Information Control in Manufacturing (2015b). doi:10.1016/j.ifacol.2015.06.200

Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.: Management of cross-domain model consistency during the development of advanced mechatronic systems. In: International Conference on Engineering Design (2009)

Giarratano, J.C., Riley, G.: Expert Systems: Principles and Programming, 2nd edn. PWS Publishing Co., Boston (1994)

Hametner, R., Kormann, B., Vogel-Heuser, B., Winkler, D., Zoitl, A. Test case generation approach for industrial automation systems. In: IEEE International Conference on Automation, Robotics and Applications (2011). doi:10.1109/ICARA.2011.6144856

Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (1996). doi:10.1145/234426.234431

Herzig, S.J.I., Paredis, C.J.J.: Bayesian reasoning over models. In: Workshop on Model-Driven Engineering, Verification, and Validation (2014). http://ceur-ws.org/Vol-1235/paper-09.pdf

Herzig, S.J.I., Qamar, A., Reichwein, A., Paredis, C.J.J.: A conceptual framework for consistency management in model-based systems engineering. In: ASME International Design Engineering Technical Conference & Computers and Information in Engineering Conference (2011)

Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. CRC Press, Boca Raton, FL, USA (2010)

International Electrotechnical Commission: Engineering Data Exchange Format for Use in Industrial Automation Systems Engineering—Automation Markup Language (2014)

Kernschmidt, K., 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 (2013). doi:10.1109/CoASE.2013.6654030

Kovalenko, O., Serral, E., Sabou, M., Ekaputra, F., Winkler, D., Biffl, S.: Automating cross-disciplinary defect detection in multi-disciplinary engineering environments. In: Janowicz, K., Schlobach, S., Lambrix, P., Hyvnen, E. (eds.) Knowledge Engineering and Knowledge Management, Lecture Notes in Computer Science, vol. 8876, pp. 238–249. Springer International Publishing (2014). doi:10.1007/978-3-319-13704-9_19

Leveson, N.G., Heimdahl, M.P.E., Reese, J.D.: Designing specification languages for process control systems: lessons learned and steps to the future? In: Software Engineering, Lecture Notes in Computer Science, vol 1687, pp. 127–146. Springer, Berlin (1999). doi:10.1007/3-540-48166-4_9

Object Management Group: MOF Model To Text Transformation Language (2008). http://www.omg.org/spec/MOFM2T/1.0/

Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (2011). http://www.omg.org/spec/QVT/1.1/

Object Management Group: Systems Modeling Language (SysML) (2012). http://www.omg.org/spec/SysML/1.3/

Object Management Group: Constraint Language (OCL), Version 2.4 (2014). http://www.omg.org/spec/OCL/2.4/

Runde, S., Fay, A.: Software support for building automation requirements engineering—an application of semantic web technologies in automation. IEEE Trans. Ind. Inf. **7**(4), 723–730 (2011). doi:10.1109/TII.2011.2166784

Strasser, T., Rooker, M., Hegny, I., Wenger, M., Zoitl, A., Ferrarini, L., Dede, A., Colla, M.: A research roadmap for model-driven design of embedded systems for automation components. In: IEEE International Conference on Industrial Informatics (2009). doi:10.1109/INDIN.2009.5195865

Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S.: Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. Technical Report TUM-AIS-TR-01-14-02, Technische Universität München (2014). https://mediatum.ub.tum.de/node?id=1208973
World Wide Web Consortium: Extensible Markup Language (XML) 1.0 (2008). http://www.w3.org/TR/xml
World Wide Web Consortium: OWL 2 Web Ontology Language Document Overview (2009). http://www.w3.org/TR/owl2-overview/
World Wide Web Consortium: SPARQL Protocol and RDF Query Language 1.1 Overview (2013). http://www.w3.org/TR/sparql11-overview/
World Wide Web Consortium: Resource Description Framework (RDF) (2014). http://www.w3.org/RDF/
World Wide Web Consortium: Shapes Constraint Language (SHACL) (2015). http://www.w3.org/TR/shacl/