Context-Sensitive Multilevel Instantiation for Space System Development

Tobias Franz

November 30, 2018

Institute of Software Engineering and Automotive Informatics
at
Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)

Dr. Christoph Seidl
Prof. Dr. Ina Schaefer

Simulation and Software Technology
at
German Aerospace Center

Philipp M. Fischer

Abstract

In model-based systems engineering, engineers from different domains collaborate on central system models. They develop abstract ideas to concrete implementations. To organize information exchange and to prevent misconceptions and conflicts, strict rules for model manipulations are required.

Multilevel modeling is a technique that can map systems development from abstract to concrete. It enables to iteratively use existing model elements to describe new ones, which are added to more concrete model levels. Editing constraints for such system models require foreseeing the following model levels. However, in systems engineering models have to be adjustable to changing requirements and projects of different complexities. As a result, depending on the application context, system models contain different abstraction levels.

This thesis presents context-sensitive multilevel modeling, which introduces a separate context model. This context can be utilized to specify context-based constraints. With, e.g., a process model as context, it is possible to specify that elements are only editable in a specific process phase. Changing requirements can be handled by updating the context model. Furthermore, context-based constraints are understandable for non-experts in modeling.

This thesis shows feasibility of context-sensitive multilevel modeling by applying it to systems engineering projects of the space domain. Besides editing constraints, these projects demand domain-specific editors and artifact generation.

Zusammenfassung

Bei modellbasierter Systementwicklung arbeiten Entwickler verschiedener Bereiche zusammen an einem zentralen Model. Dabei entwickeln sie abstrakte Ideen zu konkreten Implementierungen. Um Informationsaustausch zu koordinieren und Missverständnisse zu vermeiden sind strenge Regeln für die Bearbeitung der Modelle notwendig.

Mit Multilevel Modellierung kann die Entwicklung des sich konkretisierenden Systems abgebildet werden. Existierende Modellelemente können dabei schrittweise verwendet werden, um neue Elemente zu beschreiben, die zu konkreteren Modeleben hinzugefügt werden. Rahmenbedingungen für die Bearbeitung von solchen Systemmodellen setzten voraus, dass die nächsten Abstraktionsebenen bekannt sind. Da Modelle der Systementwicklung allerdings an sich ändernde Anforderungen oder Projekte verschiedener Komplexität angepasst werden können müssen, können die Modelle je Anwendungskontext verschiedene Abstraktionsebene enthalten.

Diese Arbeit stellt kontextbasiertes Multilevel Modellierung vor, bei der der Kontext in einem separaten Model beschrieben wird. Auf Basis dessen können Rahmenbedingungen für die Bearbeitungen des Systemmodells festgelegt werden. Mit einem Prozessmodell als Kontext ist es zum Beispiel möglich zu spezifizieren, dass bestimmte Änderungen nur in speziellen Entwicklungsphasen möglich sind. Auf sich ändernde Anforderungen kann mit einer Anpassung des Kontextes reagiert werden. Solche prozessbasierten Rahmenbedingungen sind zudem nicht nur für Modellierungsexperten verständlich.

Zur Auswertung wendet diese Arbeit kontextbasierte Multilevel Modellierung exemplarisch für Projekte in der Raumfahrt an. Neben Rahmenbedingungen für die Bearbeitung der Modelle werden dabei auch anwendungsspezifische Editoren und Generatoren vorausgesetzt.

Contents

List of Figures

1 Context and Motivation

The use of models in systems engineering is becoming more popular [1]. Models are used to organize communication between stakeholders [2] and for automatic generation of project artifacts [3]. Even software for critical environments is being generated [4]. NASA's mission to mars, 'Curiosity,' generated 75% of the source code for its lander's onboard software.

Modeling can improve both, short-term productivity as well as productivity on the long run [5]. However, the benefit of a development method depends on its integration into the development process. While model-based systems engineering (MBSE) defines basic concepts and principles [6], this work focuses on systems engineering processes for complex systems, which require multiple development phases. In particular, processes where development of later phases instantiate concepts developed in earlier phases. An example for such a process is the development of space systems.

1.1 Model-Based Systems Engineering for Space Systems

Designing and maintaining a spacecraft is divided into several phases. The European Cooperation for Space Standardization (ECSS) introduced the so called project life-cycle phases, which describe a set of engineering tasks and outputs for each phase [7]. The first phases, Phase 0, A and B start with feasibility studies and design work. The development and assembly follows in Phase C and D, the operation in Phase E and the disposal in Phase F.

Studies conducted during the early mission phases are often held in concurrent design facilities, following a defined process and making use of data models for information exchange. Information is exchanged between all life cycle phases as well as the different engineering tasks and business processes. The development of a spacecraft is an interdisciplinary process, in which engineers from various domains, such as physical-, thermal-, electrical engineering, colaborate. While current projects pass on information from previous phases via formal documents, projects exist that target supporting the whole life cycle of space projects [8]. The project Virtual Satellite follows a model-based system engineering approach where a model is used for information exchange, also between different phases. Virtual Satellite developer's vision is a system model that lasts from Phase 0 to Phase F. Model language customizations adapt the generic system model to the specific requirements of each phase.

Such an engineering process is not necessarily restricted to the development of one system, it can also support the advancement of technology. Due to the harsh environment, maintenance of space systems is costly and system bugs can result in total mission failure. To reduce the risk of system breakdown, technology used in space must have proven their robustness in previous experiments on Earth. NASA introduced the Technology Readiness Level (TRL) standard to provide a metric for evaluating the maturity of technologies [9]. The first of TLR's nine levels is *Basic principles observed and reported* and the last level is *Actual system "flight proven" through successful mission operations*. A technology's progress from TRL 1 to TRL 9 can take years and several subsequent projects,

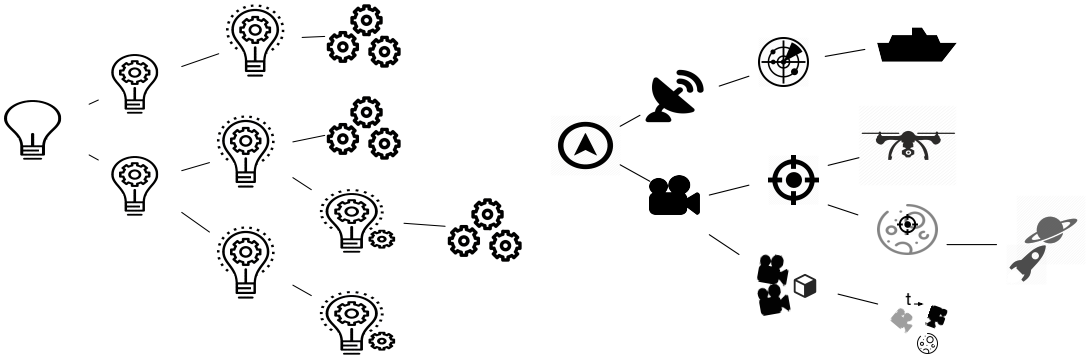

Figure 1.1: Technology's development from a generic concept to a concrete application. Each development step instantiates previous ideas to a more concrete application. The example on the right shows the idea of a navigation system.

building upon each other. The levels show that a technology develops from a generic concept to concrete application in specific missions. New projects become more concrete because they instantiate concepts developed in earlier projects. Figure 1.1 shows a possible development of a technology solution. A generic idea can result in different concepts in how to implement it. The example is based om the idea of the project autonomous terrain-based optical navigation (ATON), which develops a navigation system for spacecraft [10]. It investigates different implementation concepts, such as different sensors or algorithms. Each concept could then have different configurations (number of sensors) and on the next level different hardware setups (different prototypes). Of course not all of these development paths result in a successful application of technology and some might need more development steps than others. The example shown in Figure 1.1 shows feature tracking as one instantiation of an optical navigation system [11]. Feature tracking is a technique which extracts visual features from input images and then tracks them over multiple frames. Changed locations of features in the images can be used to reach conclusions about the observer's relative movement. Such a system can be directly used in a consumer drone, but an application in the navigation system for spacecraft needs further development. A space implementation of feature tracking might be, e.g., crater tracking. Crater tracking can use an optimized extraction of, e.g., lunar or martian craters as features. Another implementation of an optical navigation system could be a stereo camera algorithm. A space implementation could use images at different points in time; however, the project might be discarded later because of a lack of processing power.

This example of a project for the development of a navigation systems shows that system descriptions transform from generic to more concrete. Furthermore, there might be several more concrete applications for one idea. A modeling language for model-based systems engineering neets to support such a model evolution.

An important aspect in the development of systems and technology is tracing of changes [12]. Since the development period spans several years and subsequent projects, reasons for decisions and challenges might get lost. Different groups of engineers and developers might not have access to each other's internal documents and, thus, lose information how system elements changed over time. Subsequent projects might not be aware of insights gained during an earlier project. Tracing helps to monitor element's affiliation and reason of existence and it can prevent necessary elements

being absent.

A model-based approach, as targeted in Virtual Satellite, with a central model for the whole life cycle of a spacecraft, supports tracing because it automatically records changes of elements. Its links between elements of different phases connect elements throughout the complete project life cycle. As Figure 1.2 illustrates, the implementation in Virtual Satellite has model elements for different project phases that have a reference to the previous phase. The *Product Element* specifies that spacecraft can have cameras. On the next level such a camera can be configured with, e.g., the resolution. In the final assembly step, the model can contain the cameras serial id. If one wonders why an element exists, it is possible to follow the links to earlier phases and to examine the element's changes.

This work will use this camera example to consistently demonstrate different modeling concepts to allow to compare the different techniques with each other.

1.2 Challenges of Modeling in a Space System Development Process

Space system development is an interdisciplinary process. Related challenges can also be true for other domains with interdisciplinary engineering. In general modeling systems over different project and development phases leads to additional challenges: concepts that are represented as an instance in one phase may be schema data in another phase. A generic sensor in an early project phase might be instantiated as a camera in the next step. On the other hand, in the final hardware configuration, there might be different concrete cameras mounted on the space system, which are instances of the camera-concept from a higher level. Here, the camera concept is represented as an object with respect to the sensor class in the first level while it is a class for the concrete cameras with a specific serial ID on the last configuration level. Different project stakeholders see model elements differently, depending on the project phases they are related to. The above described instantiation differs from a specification process because, e.g., Phase A projects may lead to several projects of Phase B. Furthermore, one project might have several concrete configurations. A space system for autonomous navigation can have setups with different numbers of cameras with different properties and serial IDs.

While modeling the concepts of a project phase, developers might not anticipate that there will be further projects or project phases, adding more instantiation levels. An example is the follow-up project for the navigation system ATON, adding safe-mode configurations to each hardware configuration. Each assembly tree might then have several configurations containing only some of the elements defined in the original assembly tree. Each configuration can then contain an instance of the concrete camera with a specific serial id, defined on previous level.

The model structure, illustrated in Figure 1.2, can map such an instantiation hierarchy. However, having an element type per project phase diminishes modeling flexibility. The number of possible instantiations is limited to the number of element types, supported by the tool environment. In addition to that, unforeseen development complexity might not be able to be modeled or require additional modeling overhead.

Depending on the size of the project, the steps necessary to develop a system might vary. While in one project it might be sufficient to have a product, configuration and assembly model-level, an-

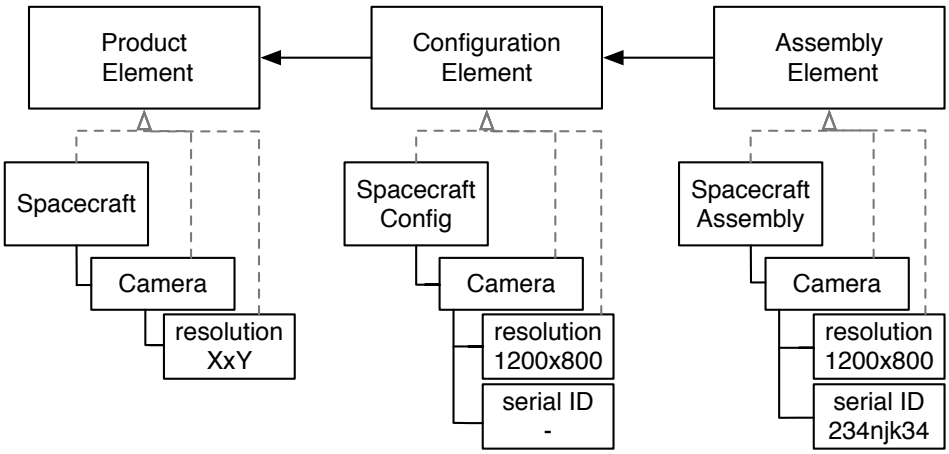

Figure 1.2: Product structure architecture with fixed number of model levels. The tool environment provides a model element per project phases. Elements of later phases reference their *type definition* in earlier phases. Properties are then copied and can be overwritten.

other, more complex project might require an additional integration step. For an infrastructure and methodology supporting the development process of space systems, it is vital to adapt to different projects to reduce modeling overhead. A tool that requires additional project steps just because of the rigid data model will not be used.

Literature illustrates a way to model concepts with different facets - class and object - in different applications. Atkinson and Kühne introduce Deep Instantiation to model this kind of instantiation in a multi-level manner [13]. To control the flexibility of this instantiation, they present the concept of *potency*. A potency defines a number, how often a concept can be instantiated. Each instantiation decreases the potency by one until it reaches zero, then the element can not be instantiated anymore. In the earlier example, the camera concept would have potency 2, the configured camera potency 1 and the concrete camera module, mounted on a prototype, with serial ID *43hjj34n3* potency 0. As Figure 1.3 shows, this approach enables modeling of different concept levels in different project phases and development steps. However, the assignment of rigid potencies to concepts restricts the flexibility to adapt the development process to varying number of development steps. A more complex project with an additional integration step would break this structure because the added step reduces all potencies so that their values cannot be set in their anticipated phase anymore. Even worse, if a project needs less steps - then tool users can change the model in unanticipated ways. Furthermore, as derived in the last section, it is in most cases impossible to foresee how often these concepts need to be instantiated.

One side effect of such a modeling approach is that users can create new model types, with new properties, at runtime. While this can be seen as benefit in some cases, it might also be a problem. In the formal engineering processes of space systems, such a high degree of flexibility is uncommon. Simple misconceptions in systems engineering and operation of space system can lead to total mission failure because of the difficult maintenance. Flexibility leaves room for misconceptions. Furthermore, dynamic type creation always limits editor capabilities because editors cannot be tailored to the specifics of each type. This contradicts the trend towards domain specific tools [14][15].

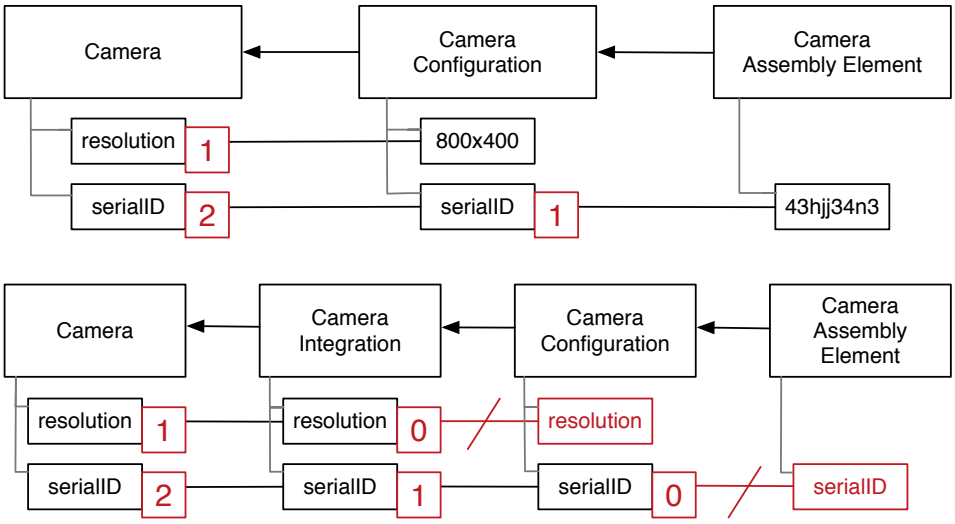

Figure 1.3: Camera satellite-element in different instantiation levels. The upper example shows potencies for a three-level process. In a more complex project with an additional integration step camera properties cannot be set in their anticipated phases anymore.

Therefore, a modeling environment not only needs to constrain type instantiation, but also the definition and extension of types to balance flexibility and tool support.

Model-driven development means the creation of a model as first step and subsequently the generation of project artifacts from it. Such an approach can help to generate large parts of the software and thereby accelerate its development process [5].

A model for the whole life cycle of a spacecraft not only covers hardware design but can also describe the onboard software of the space system. The generation of such embedded software requires to consider additional constraints. Hardware resources are in most of the cases limited and software, such as a navigation system, has additional real-time requirements [16]. To comply with existing conventions of embedded systems, generators and generated source code need to be customizable. Especially manual changes in generated code must not be overwritten by generators.

In the area of model-driven software development for space systems (MDSD), domain-specific languages (DSLs) have proven to be efficient in terms of learning effort and porject productivity [14][15][17]. To reduce the learning effort for engineers, the modeling infrastructure should support a visual customization of the concepts on different levels. This way, communication in software can be, e.g., presented in form of a UML component diagram, while communication on hardware level uses descriptions common in the hardware domain.

1.3 Goals of this Thesis

The goal of this thesis is develop concepts and a methodology to support complex, iterative systems engineering processes and to overcome the aforementioned challenges. Developed concepts should support engineering processes, consisting of several phases instantiating each other. Necessary flexibility of the modeling tools required by this approach should be controllable and there should exist methods to restrict model manipulations and concept instantiations.

To efficiently implement a modeling environment for space systems engineering, a requirements

elicitation is necessary. The following list discusses requirements for formal engineering processes such as described before.

REQ.1 **The environment shall support dynamic type instantiation.** The motivation of this work is to handle and reduce modeling complexity caused by engineering tasks that work with elements on different conceptual levels. Elements that need to be used on different locations in a model — modeled separately each time. Mapping complex, multilevel contents to a two-level modeling environment not only increases modeling effort, but also disregards their logical affiliation. As Section 1.1 shows, the development of technology and systems iteratively instantiates earlier concepts and refines them towards a more concrete application. To improve model-based systems engineering, models need to be able to map this affiliation. In concrete, it is necessary that elements in a system's data model can have instance-relationships to elements in another phase.

REQ.2 **The environment shall enforce model manipulation constraints.** Enabling multilevel modeling, as demanded in Requirement REQ.1, brings a high degree of flexibility to the model and its tools. To prevent misconceptions, the environment needs to enforce rules and constraints to control this mechanism. Especially, setting modeled element parameters needs to be restricted. Using - instantiating - the implementation of a network protocol in a satellite makes it necessary to configure the connection parameters, how elements are connected, while implementation parameters, such as the signal voltage, cannot be changed anymore. Besides editing of parameters, also their visibility and wheather an element can be instantiated at all shall be limited.

REQ.3 **The environment shall be adaptable to different complex projects.** The development of technology and systems is strongly varying. Depending on the complexity of the application domain, projects might require more development time and steps than others. Even though optical navigation with, e.g., feature tracking can be used in a simple consumer drone and in an autonomous spacecraft, their development time will be highly different. A feature tracking for spacecraft might instantiate the concepts and optimize them for, e.g., to a crater tracking that can be used on foreign planets. For modeling environments that means, that there shall not be a fixed number of element instantiations and, thus, also not a dedicated element per phase. Because in most cases it is impossible to anticipate, how often a concept needs to be instantiated, simple numbers shall not be used to constrain the instantiation process.

REQ.4 **The environment shall allow interdisciplinary development.** Model-based systems engineering is an interdisciplinary process, different groups of engineers work on the same data model. Editors and views on the model should only contain relevant information for the current user. To organize such a process, the data model and its environment need to implement the principle of separation of concerns and handle overlapping between the different domains. Enabling multilevel concept instantiation must not restrict these basic principles.

REQ.5 **The environment shall allow domain-specific representation and editors.** Domain-specific languages significantly reduce learning effort and prevent misconceptions[18]. An interdis-

ciplinary process, such as development of spacecraft or cars, does not work with general purpose languages and generic editors for all parts of the model. Editors that are not understood by engineers will not be used properly. But not only the input of data needs to be domain-specific, some domains might require data visualizations, such as diagrams or simulations of the system with, e.g., a presentation of the mass distribution. Multilevel editors shall not be restricted to generic concepts, such as boxes and lines.

REQ.6 **The environment shall allow artifact generation from the model.** As Atkinson and Kühne present, a development process can be significantly accelerated when using model-driven engineering [5]. An interdisciplinary system model with data from different engineers can be he highly useful source for source code generation. Furthermore, generation of documentation from the model can improve communication between different groups of engineers and stakeholders, because these generated documents can be kept up-to-date. When changing an interface, engineers have to update hardware, source code and several documents. Forgetting to update one of these locations leads to inconsistency and, thus, errors. With model-driven engineering, changes only need to done in the model. Documentation, hardware and software are updated automatically. Multilevel models must not restrict artifact generation.

REQ.7 **The environment shall restrict dynamic type extensions.** Developing systems, such as satellites, is a process that has been done before. Even if there are new parts, prototypes or technologies involved, system development does not completely reinvent all concepts. In engineering processes, such as concurrent engineering facilities, domain-engineers have to focus on their domain rather than how to add new properties to a system element. Dynamically extending the data model, as possible with multilevel modeling, should follow strict rules. Otherwise extended elements can neither be used for communication between different groups nor for generation from the model and, thus, are useless.

While most existing multilevel-based tools in literature have been evaluated using theoretical examples (e.g. [19]), this thesis demonstrates its concepts with a proof-of-concept editor and an exemplary application of the concepts on actual space projects. To prove not only theoretical usefulness, concepts should be developed for generic applications, but the implementation should be as part of a concrete tool for space systems engineering.

The remainder of this thesis is structured as follows: the next chapter presents relevant literature. The third chapter introduces related work. The fourth chapter specifies requirements for multilevel-based systems engineering process and then develops context-sensitive concepts for such a development process. Chapter 5 presents methods how these modeling concepts can be applied in common modeling environments. A concrete implementation of the developed concepts for the space-engineering domain is presented in Chapter 6. Chapter 7 evaluates the concepts and proof-of-concept implementation by applying them to challenges of systems engineering. Chapter 8 discusses the results and the last chapter gives a conclusion.

2 Background of Model-Based Engineering

This chapter introduces necessary technologies and methodologies for model-based engineering. The first section introduces model-based systems engineering in general. Section 2.2 presents a special type of modeling, *multilevel modeling*, which can be used to describe conceptual *instance of* relationships. The last section of this chapter presents model-driven software development and highlights its increased productivity for engineering processes.

2.1 Model-Based Systems Engineering

System engineering is an interdisciplinary approach to develop and implement complex, technical systems [20]. Model-based systems engineering is a collection of process methods that support systems engineering by using models [2]. Models can formalize such systems engineering practices. MBSE tries to replace the document centric approach, used in classical systems engineering, with models to improve communication [21]. Models substitute documents for specifications, interface requirements, test plans and analysis.

2.1.1 Models in Systems Engineering

Models provide abstraction of a system or software and allow to focus only on relevant details [22]. System models ignore extraneous details and, thus, help all kinds of engineers to understand real world systems. Models can help to predict system qualities, risks and costs and reveal the impact of system changes. Furthermore, models can improve communication between stakeholders.

Models are either used as precursor / prototype for the development and implementation of a system, or they are derived from an existing system to understand its purpose or behavior [23]. Figure 2.1 shows the different types of models. Models can either be a constructive engineering model, such as the satellite model, or they can help to understand existing systems, such as the planet Mars, and thereby help to develop other systems.

2.1.2 Model-Based Principles

Model-based engineering is about adding models to engineering processes to support specification, design, integration, validation and operation of a system [2]. MBSE targets to collect system related information in central models and provide relevant parts of this information to corresponding system stakeholders. Models can exist several project phases and dynamically support life-cycle management and tracing of changes [8][26]. Tools can support information exchange by persisting models, managing access rights and presenting their content in different ways. According to Estefan, activities that support the engineering process are supported by increasingly detailed models [2]. The authors point out that it is important to know for stakeholders that the model grows

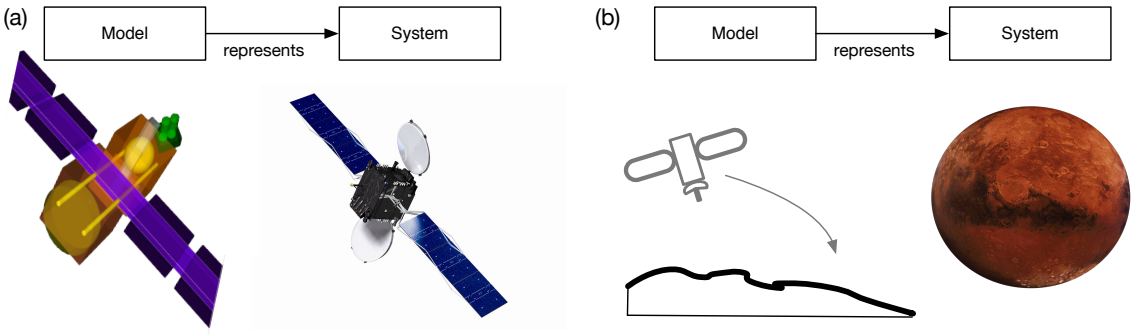

Figure 2.1: Different model types for engineering. The left model (a) is an engineering model, created with Virtual Satellite, that supports developing satellites [24]. The right model (b) represents Mars, it can help to find a landing spot for spacecraft. Image: [25]

from a generic, fidelity viewpoint to a sufficiently faithful model for compliance assessment.

Baker et al. argue, during the engineering process, models develop to a cohesive, unambiguous representation of a system or process [27]. Validation activities check the model's correctness, verification can be used to evaluate if the system is evolving as planned in the model. Requirements and design are iteratively adjusted until completeness and quality criteria are satisfied. With the replacement of documents against models as information repository, most engineering activities can be automated. According to Baker et al. reviews can be automated by interrogating the model rather than to read and interpret documents [27]. Also verification can be automated, validation can be done by using the models, e.g., in the customers context. Baker et al. also mention, that traceability is integral with models [27].

2.1.3 Model Evolution in Systems Engineering

If information in models is persisted over multiple development steps, it is necessary to trace system characteristics to support software management, evolution and validation [26]. Figure 2.2 presents Galvão and Goknil's understanding of model refinement [26]. According to them, a model is a symbolic system represented in an appropriate language for a given purpose. Each development step means a refinement of the model. Compliant to the presentation of concept development in Section 1.1, the authors argue that more abstract models are transformed to more detailed, concrete ones. Galvão and Goknil describe system development as a series of model transformations and argue that changes in one model have to be propagated through the rest. To archive this, they present a set of traceability methods, which help to trace system characteristics through different models.

Paige et al. analyze several applications of MBSE and investigate model evolution [28]. They distinguish between MBSE applications where metamodels change because of model evolution and where it stays in the initial state. With *model-metamodel co-evolution*, changes in a metamodel require to update the instance model because it also means changes in the model serialization. Paige et al. highlight the challenges of model migration to an updated metamodel [28].

Developers of the project Virtual Satellite work on a related but different approach where the base metamodel does not need to evolve [8]. As part of the German Aerospace Center (DLR), they develop

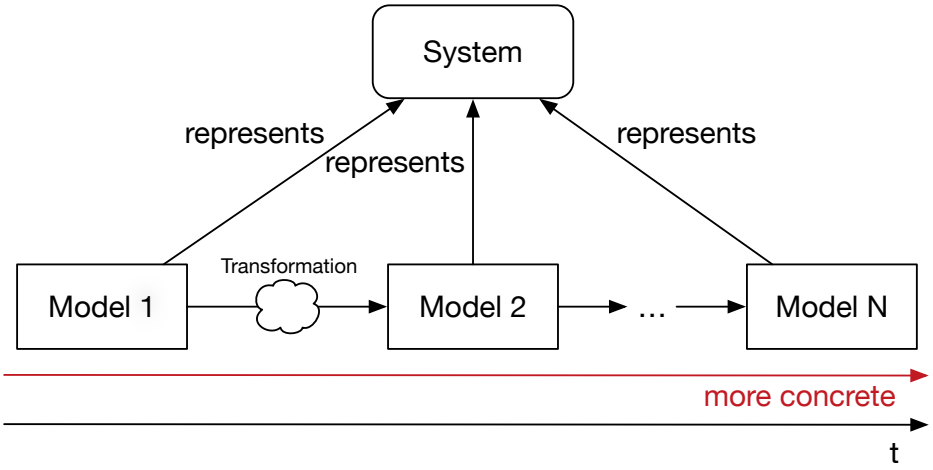

Figure 2.2: Galvão and Goknil's description of model refinement in model-based engineering [26]. Transforming abstract system models into more conrete ones.

model-based methods for the development of space systems. As Figure 2.3 presents, the goal of this project is to support the whole life-cycle of a spacecraft by keeping one model from design phase to disposal. While different project phases are supported by specialized language extension, the conceptual base remains the same in all steps. Here, the model does not only support communication between different stakeholders of a project but encourages also information exchange between different project phases. This way, changes of the design of a spacecraft are tracked over its complete life-cycle and its progress can be analyzed better. Spacecraft operators can then, e.g., comprehend, why a decision in the design phase was made and use the spacecraft accordingly.

To implement phase-specific extensions, Virtual Satellite introduced an extension mechanism that is based on the linguistic metamodel and generates domain-specific infrastructure. As the instance of these extensions are linguistic instance of the static metamodel, it is possible to introduce new *Concepts*, extension element containers, without changing the linguistic metamodel.

Maintaining one data model for the whole life-cycle of a product brings additional challenges for the underlying conceptual language. Transformations from abstract to concrete, as Galvão and Goknil describe [26], have to be mapped in one model. As Figure 2.3 shows, a reference type for references between model elements of different phases has to be found so that the transformation from abstract to concrete can be represented. A modeling technique that handles such transformations from abstract to concrete is multilevel modeling.

2.2 Multilevel Modeling

Mapping complex systems and their development process into models is challenging. The mainstream *"two level" modeling paradigm* - with a metamodel, defining the modeling language, and an instance model - cannot map most conceptual system information properly [29]. This mismatch makes models and their editors more complex. Following the general engineering principle to minimize accidental complexity, Atkinson and Kühne present modeling techniques to better represent multiple levels of abstraction in conceptual information. Their goal is to decrease redundancy in

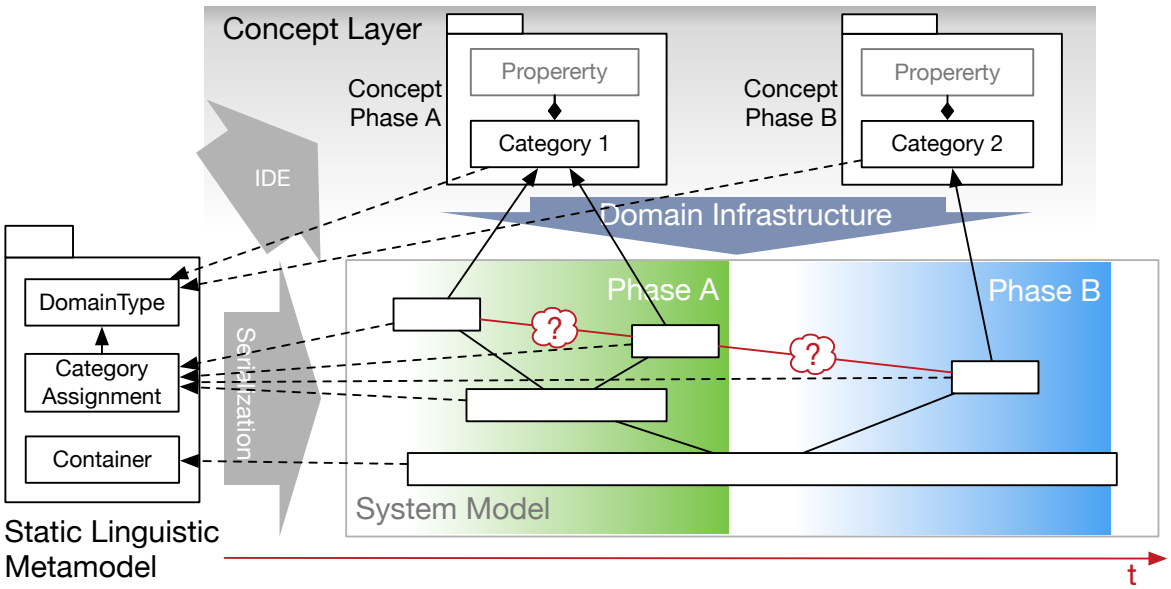

Figure 2.3: Model-based system engineering for the whole life-cycle of a system [8]. One model is used for design phases until disposal. There can be extensions specific for some phases but the linguistic base remains the same.

models and to increase flexibility. As Frank points out [30], a restriction of our communication in everyday life to primitive concepts such as class and object would be regarded as completely unreasonable. Languages provide concepts that enable communication with previously defined elements, so that we do not have to explain everything from scratch. To move beyond the "'two level"' paradigm, Atkinson and Kühne argue, modeling concepts are necessary that can be applied in a uniform way across all levels of a classification hierarchy [29]. They introduce elements, which can be seen as class and as object, depending on the context.

Following these ideas, modeling software can be optimized [31]. The model of a system with multiple cameras can be simplified by first creating a camera type with the camera's parameters and subsequently instantiate the new element in the system description. This not only simplifies the modeling process but also reduces the redundant definition of the camera's parameters.

2.2.1 Orthogonal Classification Architecture

The orthogonal classification architecture (OCA) explains how multiple levels, as mentioned before, can be implemented using "two level" infrastructures [32]. The architecture separates classifications into two independent dimensions. The first dimension represents the classical, linguistic levels which define the statical parts of the modeling language and, thus, its serialization and infrastructure. The second, independent dimension is conceptional. This dimension can consist of an arbitrary number of levels. All conceptual classification levels can be represented in the same way and, thus, support deep modeling, meaning that instances can be instantiated again.

Figure 2.4 shows the two independent dimensions. In linguistic sense, metamodels can be defined using the Meta Object Facility (MOF). Types created in a metamodel, such as, e.g., *Object* can then have an instance on the next linguistic level. The *UVCamera1* in the instance model is an object with

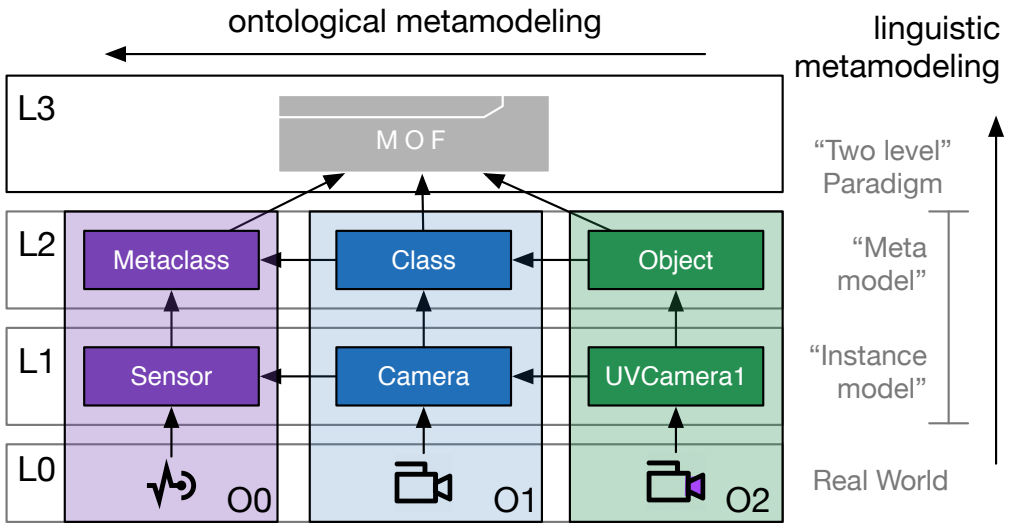

Figure 2.4: The Ortogonal Classification Architecture. Linguistic metamodeling defines the static parts of the modeling language of lower level. Ontological metamodeling is conceptional and has to be realized dynamically.

respect to the linguistic metamodel. On the other hand it represents a camera in the real world. The *UVCamera1* object is serialized as defined on level L2. Independent from this language-definition dimension, there can exist arbitrary classification relations on one linguistic level. The *UVCamera1* is a *Camera*. A *Camera* is a *Sensor*. Besides cameras there could also exist other kinds of sensors on this model level. The linguistic metamodel defines the number of conceptual levels that can exist in the linguistic instance model.

2.2.2 Terminology

To clarify different terminology used in literature, this section lists key terms of multilevel modeling. While presenting basic multilevel concepts, Aktinson and Kühne also introduced first terms for their concepts [13].

Level If not mentioned otherwise, *level* always refers to the ontological dimension of the OCA, introduced in Section 2.2.1. Such a level contains elements at one conceptual classification level.

Model The whole collection of elements on all levels. In later literature sometimes referred to as *Ontology*, with the term *model* as the set of elements only at one classification level - here referred as level [19]. However, the term *model* is used widely in all kinds of domains and depending on the context, it might also refer to the general concept, defined in Section 2.1.1, or to a technical entity.

Clabject (Class + Object) The duel-faced concept, that can be type or instance, depending on the context.

Field *Ontological attributes* of clabjects are fields. A field is a triple of name, data type and value.

Potency Linguistic attribute that specifies how often a clabject can be instantiated. The potency decreases with each classification level in which a clabject is instantiated until it is zero. If a clabject's potency is zero, then the clabject cannot have any further instances.

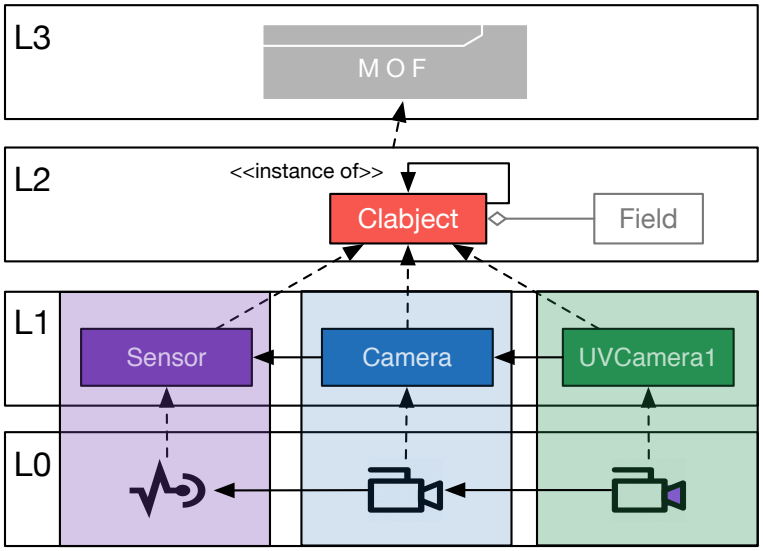

Figure 2.5: The linguistic metamodel enables an arbitrary number of conceptual levels by defining an element that can be type and instance at the same time. All domain elements, such as *Sensor*, *Camera* and *UVCamera1* are a linguistic instance of this, so called, clabject.

Later, the community refined this terminology and added new terms to reduce potential for confusion [19]. The following presents further terms, relevant for thesis:

Trait A *linguistic attribute* is called trait. To clearly differentiate, the term *attribute* - without specification if linguistic or ontological - then usually refers to fields.

Durability Initially, the linguistic attribute that defines how long a field can be instantiated was also called potency. To reflect their less important role, potencies of features are now called durability.

Level numbering The original numbering scheme for classification levels, presented by the Object Management Group, starts with number three for the most abstract level and decreases towards the most concrete level. Since this scheme restricts the number of levels, it is not appropriate for ontological classification with potentially arbitrary number of levels. Following Kennel, we start with zero for the most abstract level and higher labels for more concrete levels [19].

2.2.3 Implementation Patterns

Based on the OCA, Aktinson and Kühne presented potency-based multilevel modeling (Sometimes also called deep meta-modeling) [13]. This pattern enables an arbitrary number of levels by using the conceptual dimension of the previously described OCA. As Figure 2.5 shows, in the centre of the pattern are elements with a dual facet: They are instances with respect to an element of higher level and type for elements on lower level. Thus, interpreting such a clabject is *context dependent*.

Because this kind of modeling spans several meta-levels, Aktinson and Kühne argue that it is necessary to control the instantiation depth of elements and features. As Figure 2.6 shows, a positive number can be attached to clabjects and its features to specify the number of levels in which the element can be instantiated. The potency decreases automatically in lower meta-levels. Once its value is zero, the element cannot be instantiated anymore. In Figure 2.6 the *outputType* reference

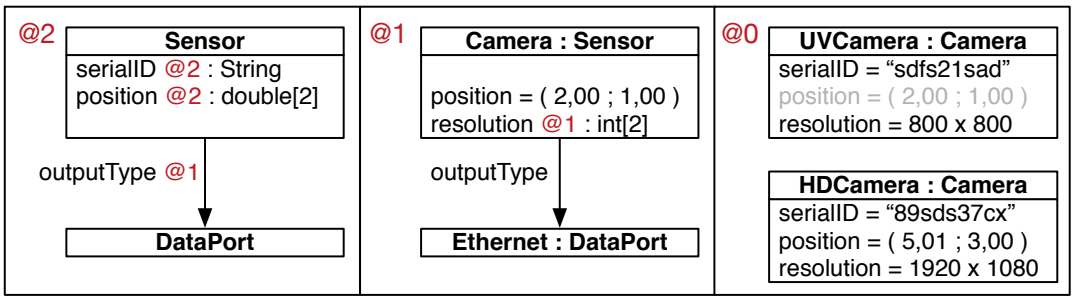

Figure 2.6: Potency-based multilevel modeling, using the syntax proposed by de Lara and Guerra [31]. The sensor's features *serialID* and *position* can be changed in the lower two levels, while the *outputType* has to be sez on the next level.

can only be set in the middle meta-level because its potency is "1". The middle meta-level's camera element can define a default value of the position attribute, which can be overwritten in the lower level.

Since most modeling environments do not natively support multilevel modeling, de Lara et al. presented a set of patterns for implementing such concepts [31]. The description of the patterns focuses on use cases where multiple levels are implicit and compares different implementation techniques. De Lara et al. introduce patterns for dynamic type and feature creation, reference configuration and element classification. For all these patterns they compare potency-based multilevel modeling with classic implementation techniques. These techniques are static types, explicit dynamic typing, promotion and, the Unified Modeling Language (UML) related techniques, Powertype and Stereotype. While explicitly modeling dynamic types has a similar level of flexibility as potency-based multilevel modeling, it always introduces accidental complexity. The other implementation techniques do not support all of the multilevel model concepts.

This discussion, comparing potency-based multilevel modeling with classical modeling techniques, can answer the question if multilevel modeling makes sense under given circumstances. De Lara et al. conclude that if an environment needs to handle dynamic type and feature creation and classification with more than two levels, multilevel modeling offers the most flexibility and simultaneously removes accidental complexity. Considering Requirement REQ.1, this suggests, that multilevel modeling techniques can benefit model-based systems engineering environments, as described in Section 1.1.

Neumayr et al. go one step further by comparing different techniques for explicitly modeling multiple levels [33]. Their comparison criteria are model compactness, the flexibility for querying elements within a model, how easy new abstraction levels can be added to a domain and how well relationships are handled over different abstraction levels. Their result is that potency-based multilevel modeling supports model compactness and querying flexibility best but adding new abstraction levels requires to change potencies on higher levels. To overcome this problem, Neumayr et al. introduce *multilevel objects* which have a similar structure as potency-based multilevel modeling but a different mechanism to control the multilevel modeling's flexibility.

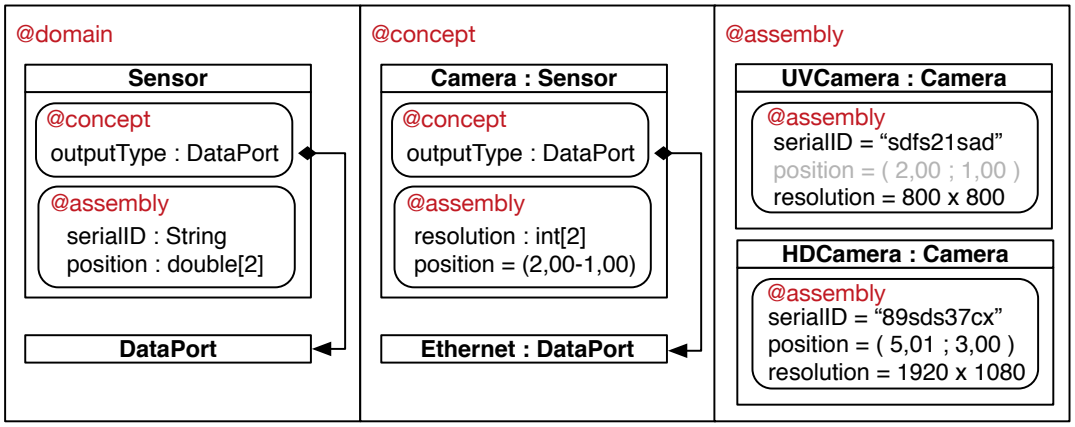

Figure 2.7: Modeling with multilevel-objects. In contrast to Figure 2.6, attributes are encapsulated in abstraction layers within the objects. Because targeted abstraction layers are referenced, it is possible to add new layers in between without having to change the control mechanism in other layers.

2.2.4 Control Mechanisms for Deep Modeling Flexibility

As mentioned before, Aktinson and Kühne introduced the concept of *Potency* to control flexibility arising by multiple meta-levels [13]. A positive number defines in how many levels a concept can be instantiated. As argued in Section 1.2 and by Neumayr et al., this concept is simple and easy to understand but it prevents from adding new abstraction layers to an existing domain [33]. A different concept is used by *multilevel-objects*, also called *m-objects*, which use a mechanism of encapsulating abstraction levels within an object [34]. Features of such a multilevel object are then mapped to abstraction levels, which are in a linear order from most abstract to most concrete. In contrast to assignment of potencies, it is possible to add independent abstraction levels later, because new abstraction levels can be integrated into the existing linear order.

Figure 2.7 shows how m-objects look like. Each object has encapsulated abstraction layers with attributes in it. With this approach, it would be possible to add a configuration layer between concept and assembly, without breaking the current structure. In contrast to potency-based multilevel modeling, *serialID* and *position* would then still be editable in the assembly layer. Fischer demonstrated the problems of potencies for systems engineering and suggested a similar approach, which he called it context-aware potencies [35].

2.2.5 Model Presentation

Linguistic metamodels are used as basis to generate a model's infrastructure. In the classical two-level paradigm, all types are defined statically and, thus, an element's representation can be specified as part of this model implementation. In a multilevel environment, where types can be created dynamically, this does not work. Since domain elements are defined in the ontological dimension of the OCA, there is no infrastructure derived from their metamodel. A model element's representation has to be chosen dynamically. Atkinson and Gerbig presented a visualization search algorithm, which tries to find an appropriate dynamic representation and if none is available it falls back to the linguistic one [36]. The first step of the algorithm is to search for a visualization in the inheritance tree. If no super type has a visualization, then the algorithm searches in the classification tree.

Volz et al. plan to use a meta modeling language to define dynamic diagram customizations [37]. Besides the application models, a set of language definition models defines the element representation of the domain elements on the next levels. Such a language can be simple and describe only icons for each type but it could also be as powerful as defining how a editor's user interface looks like, which features are visible and which not or define a concrete textual syntax for the domain models.

Viyović et al. present a tool where complete graphical domain-specific languages can be designed dynamically [38]. Such a tool can be used to specify how elements on the next level are visualized. As domain information and layout data should be strictly separated [39], data about how the next level is visualized should not be mixed with domain-information about the system.

2.2.6 The Instantiation Transformation

According to before mentioned literature, the key difference between multilevel modeling and the classical "two level" paradigm is that there exist elements which can be type and instance at the same time and, thus, an arbitrary number of classification levels. This new possibility of concept instantiation over multiple levels highlights the importance of this transformation process. Kennel argues that in constructive modeling, the instantiation operation is the most important one [19]. According to him, if the most abstract model is sufficient, the instantiate operation is the only creational operation necessary. This *creational power* has to be considered carefully. While it is straightforward for traits and fields, Kennel highlights the complexity for references and connections. The instantiation of a clabject with a reference to another element would introduce further new elements to the model. Thus, a global - level wide - conformance to the type-model is hardly possible and might require a complete instantiation of the higher level model. Because of that, Kennel argues, that the instantiation operation should ensure that it does not negate the type-instance relation, rather than ensuring *global conformance*. According to him, it might be possible to guess probable participants of a connection but to avoid automatic creation of artifacts, clabjects should not be connected. He also mentions, that *Multiplicity Satisfaction* is challenging. After a complete instantiation of the high-level model's clabjects, there are as many clabjects in the classified model as in the classifying model. He argues that a tool cannot decide how many clabjects to create for a reference or connection.

While multilevel modeling simplifies user models and reduces accidental complexity [40], the model environment has to enable these achievements. The paradigm shift from "two level" modeling to models with an arbitrary number of model levels, with domain types created at runtime, increases the need for dynamic modeling environments. Classical "two level" modeling could provide tools, serialization and as mentioned in Section 2.2.5, visualization statically by generating infrastructures from type definitions. Multilevel environments have to handle these aspects dynamically, because domain concepts are defined at runtime.

For an implementation of a multilevel modeling environment, one has to consider the creational power of the instantiate operation in combination with the dynamic nature of this type of modeling. Object creation of shared, dynamic behavior can be implemented using prototypes [41].

Relation to Object Creation With Prototypes Object creation with prototypes can be used following design patterns [42] or with language features, as implemented, e.g., in JavaScript [43]. As with multilevel modeling, the creation of new domain elements can be done dynamically. Furthermore, similar to the instantiate operation in multilevel modeling, object creation from other concepts

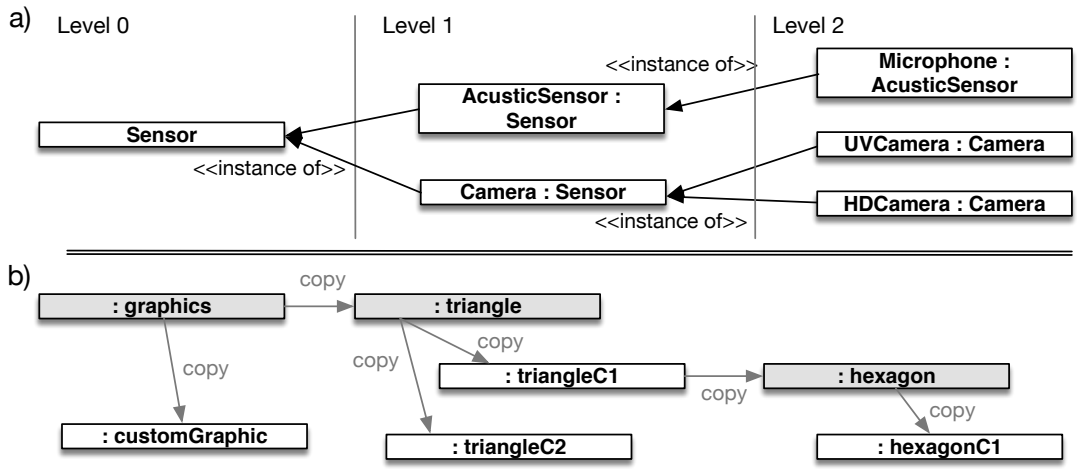

Figure 2.8: Conceptual difference of multilevel modeling (a) and object creation from prototypes (b). Multilevel model elements can be instantiated in next classification levels only. Prototypes can always be manipulated and copied.

can be done incrementally. Object creation is done by copying exemplary prototypes rather than instantiating classes [44]. Changing a type means modifying the dynamic prototype, rather than the static class definition.

Conceptually, however, multilevel modeling and object creation by prototype, are different. Multilevel modeling introduces an arbitrary number of classification levels, prototyping tries to avoid classification where possible [41]. Object creation by copying from a prototype cuts the relation of both objects [44], clabjects keep being an instance of another clabject. Prototypes' properties are described with exemplary values; Clabjects' fields, as class attributes, by abstraction [44]. Prototypes can always be extended with new fields and behavior, clabjects only when added to their type on higher classification level. Considering Requirements REQ.2 and REQ.7, object creation with prototypes allows too much flexibility. Furthermore, classification, as in multilevel modeling, can be used to model the evolution from abstract to concrete in systems engineering, as presented in Section 2.1.3.

Figure 2.8 shows the conceptual differences of multilevel modeling and object creation with prototypes. The sensor-camera example demonstrates the classification levels and the model elements' *instance of* relations. In contrast, all prototypes and objects created from these prototypes live in the same classification level. After object creation, there is no connection or reference between the new object and its prototype anymore. Prototypes are a useful creation pattern for graphical tools, where elements need to be manipulated dynamically. The visualized elements can then be described and edited without the need of abstraction, as needed with, e.g., classes. Since the prototype-based object creation is more flexible, it is possible to create an object structure, such as shown in the multilevel example (a) also with prototypes. One could create a sensor object, copy this and customize it to a camera element and so on.

Although multilevel modeling and object creation with prototypes are fundamentally different, both concepts share the motivation to overcome the problems of two statical levels - class / object and metamodel / model. Both creational methods, copying from a prototype and instantiating a clabject, produce a dynamic object from another dynamic object. When implementing a multi-

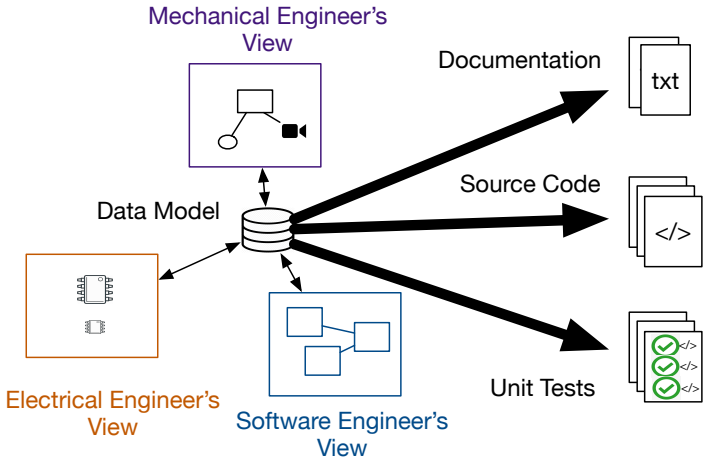

Figure 2.9: Model-driven software development. Model-driven means that a model is created as first step. This model is then used to generate source code, documentation and test code. The model creation can be done by using editors for different system stakeholders.

level modeling environment, the arbitrary number of conceptual levels has to be mapped to the two linguistic levels. The linguistic metamodel, defining the dynamic multilevel runtime, and the environment around it have to implement how the instantiation operation as well as the model presentation and editors work. This transformation of multiple conceptual levels to the one dynamic software level suggests to use aspects of the prototype paradigm to implement multilevel modeling. The first step of instantiating a clabject can be implemented by copying it from one level to another.

Besides creational aspects of the initialize operation, Neumayr and Schrefl point out that the instantiation process needs to resolve conflicts between different possible inheritance aspects of clabject relationships [45]. In particular, they consider *Dual Deep Instantiation* where clabjects can have parents considering both, the type-instance relation and the generalization relation. Clabjects inherit properties from their *class-clabject* on higher instantiation level as well as their *superclabject* on higher generalization level.

2.3 Model-Driven Artifact Generation

Artifact generation from models is widely used in industry [46]. Modeling techniques that support engineering processes should be applicable for such a generation. While model-driven development is not formally defined, the common understanding is the process of describing a system or software in a formal model and then generating project artifacts from it [22]. Due to many relevant aspects of a system, engineers may use different notations and views on the model. Model-driven software development describes the process of generating source code, unit tests and documentation from the model. Main motivation for model-driven development is to increase productivity [5]. Productivity benefits twice from this kind of development: Short-term productivity increases, because developers can use the model to generate new features from it. Long-term productivity rises because changes in requirements can be handled by changing the model and regenerating, rather then reimplementing the software manually.

Figure 2.9 shows the principles of model-driven software development. Different developers and engineers of the system model see their relevant parts in specially for them created views. Subsequently, the model is used to generate software and documentation from it.

Empirical analysis shows that model-driven engineering improves controlling and communication [46]. Stakeholders exchange more organizational knowledge and react faster on changing requirements. Hutchinson et al. present that most projects with model-driven approaches use the Unified Modeling Language (UML). However, more than 40% of MDSD users think that UML is too complex. The study also presents that most failed model-driven projects did so because of a lack of training and knowledge about the modeling language. Another survey about UML's extension mechanism shows a decline of applications of UML in the years before the study appeared (2010) [14]. The authors suggest that new tools and environments for creating domain-specific languages (DSL) leads to a trend to create modeling languages from scratch. While code generation and increased communication between stakeholders improves the general productivity of products, general purpose languages, such as UML, require too much learning effort and leave room for misconceptions [15]. Research shows the importance of domain-specific representations of the model elements. Domain-specific languages that are developed from scratch only contain concepts that are needed in the domain and, thus, improve their comprehensibility.

3 Related Work

Techniques and methodologies presented in the last chapter are already used in industry and research. This chapter presents applications of model-based engineering in the space domain and presents related work of multilevel modeling. The last part of this chapter presents projects, where models are not only used for information exchange but also for artifact generation.

3.1 Model-based Systems Engineering

MBSE is widely used in industry. An example in the space domain is the CubeSat Reference Architecture [47]. The CubeSat project aims to simplify small satellite missions by using off-the-shelf components. It, therefore, presents a reference model that can be used as a guide for development of the satellite. It uses the Systems Modeling Language (SysML) and it covers costs, requirements and life cycle aspects. This application of model-based systems engineering shows that models can be used efficiently to document systems and their usage. The CubeSat model provides an easier and deeper understanding of system aspects - necessary for developing such a satellite. Costs and requirements in the model can support development effort estimations and, thus, improve project management and planning.

Another example for an application of MBSE in the area of interface management is presented by Vipavetz et al. as part of the National Aeronautics and Space Administration (NASA) [48]. Vipavetz et al. introduce a seven-step process to maintain interfaces. The process starts with identifying interfaces and ends with the integration of system components. In each step, the model is used to either add or read information. The model thereby manages and supports communication between different parties and persists information about decisions and changes of the system. Managing model manipulations, such as described in Requirement REQ.2, should be adaptable to such processes.

3.2 Multilevel Modeling Environments

Multilevel modeling has become a popular research area in recent years. Universities and research institutes developed multilevel environments to refine concepts and develop practices how multilevel modeling can be used in industry.

On basis of their multilevel foundations, Atkinson and Gerbig presented an environment that not only enabled basic multilevel modeling, but also enabled domain specific, textual and graphical modeling languages [49]. To visualize elements in their environment, called Melanie, they differentiate between two kinds of editors: *generic editors*, for all multilevel models, and editors based on *domain-specific* languages. While multilevel modeling enables an arbitrary number of conceptual model levels, they all share the same linguistic metamodel. As presented in Section 2.2.1, the linguistic metamodel is basis for the environment of all application models. The concepts defined in the linguistic metamodel, such as the *Clabject*, can be used to create a general-purpose editor. Domain-specific editors can be created by specifying a domain-specific representation of the domain elements, as described in Section 2.2.5. The tool described by Atkinson and Gerbig supports a

domain specific definition of a concrete syntax in graphical and textual form. Because of the drawbacks of a compiler-based textual model editor, such as grammar limitations by the model-based parser generators, the tool uses an editor based on model projections. While Melanie demonstrates technologies with high potential, the tool's capabilities do not last for an application in model-based systems engineering. Potencies are too rigid and, thus, not feasible for modeling with changing requirements of systems development. The tool, thus, does not fulfill Requirement REQ.3. Furthermore, while the tool enables dynamic customization of the concrete syntax of the editors, these capabilities are restricted. A complete redefinition of the grammar of diagrams and textual languages is not possible and, thus, Requirement REQ.5 is only partly fulfilled.

Another multilevel modeling environment that shares a lot of common ideas and concepts with Melanie is MetaDepth [50]. MetaDepth is a textual modeling tool that focuses on software development. It is build on the tool suite for the transformation language Epsilon. The authors of both environments presented an in-depth comparison highlighting the difference of both tools [51]. MetaDepth also uses potencies and the tool is focused on software development rather than systems engineering in general.

Another example implementing this kind of architecture is the Open Meta Modeling Environment (OMME) [52]. To encourage creativity, this tool tries to achieve a more flexible modeling in the first part of a systems development process. Outcomes of this first informal modeling step are then connected to formal models, developed in later project phases. The idea of this environment is to enable developers to create concepts on-the-fly, before thinking about how to classify the element. Users can then instantiate concepts created in this tool to develop more concrete models or transform it to a more abstract element that can be instantiated on same level. To enable elements having types and instances, their entities correspond to the clabject, described earlier. The tool tries to use neutral naming because terms such as "class", "object" or "clabject" are sometimes not well understood by domain experts. Besides the application modeling environment, the authors introduce two additional languages to define domain-specific representations and constraints for the domain elements [52]. While their concept is interesting for creative and flexible domains, it is unsuitable for space engineering. As mentioned in Section 1.1, MBSE for space does not allow too much flexibility for domain users. The tool, thus, does not fulfill Requirement REQ.7 sufficiently because it targets a different user group.

Frank presents another multilevel modeling environment that focuses on hierarchical language development [30]. Their motivation is to use multilevel modeling to fix that models are extensively used in early conceptual phases, but later changes are only applied to source code. Frank proposes to use multilevel modeling to better represent different abstraction layers of domains and organizations. A reference language can be used to model a complete domain, technical reference languages can be used in whole organizations and more concrete, technical languages can be applied project- or department-specific. This way, models can be used and updated for the whole development process of a system. The different abstraction layers of models are also well suited for system integration, because compatibility of components can be checked on all levels.

3.3 Model-Driven Artifact Generation

Model-driven software development is widely used in industry, even critical space projects use code generation [4]. NASA's lander mission to Mars, *Curiosity*, used about 75% of generated code from

models. Also the European space agency (ESA) develops a tool for model-driven software development for space systems [53]. Their tool-chain, called *The ASSERT Set of Tools for Engineering* (TASTE), is based on complementary languages to cover system architecture and data descriptions in a model that can then be used to generate source code for embedded systems. TASTE is not only generating source code, but also build system files and documentation.

Another space project that used model-driven software development intensively is the project *matter-wave interferometry in microgravity* (MAIUS) [18]. The project developed a sophisticated set of interwoven domain-specific languages to generate the onboard software code for a sounding rocket. MAIUS executed complex physical experiments in space and therefore needed an highly interdisciplinary project team. Due to its electronic components were experimental, also drivers for the hardware had to be developed as part of the project. Furthermore, in parallel to the hardware development physicians planned the sequence of experiment steps. Because neither physicians nor the electrical engineers were experts in software development, the creation of drivers and software for the experiment execution was challenging. To handle the frequently changing requirements and hardware interfaces, a software engineering team developed a model-driven environment as shown in Figure 2.9. Electrical engineers described the developed hardware in domain-specific languages, physicians modeled the experiment sequence in graphical languages similar to project-specific, extended activity diagrams. Software engineers then developed code generators, which used the electrical descriptions to generate driver code and the experiment sequence description to generate the rest of the experiment control software. This way, engineers communicated via the model and then most of the rocket's onboard software could be generated automatically.

4 Definition of Context-Sensitive Multilevel Modeling

In model-based systems engineering, models evolve from the description of abstract concepts to concrete implementations. Multilevel modeling is a method that enables to model concretization of elements step by step. Using multilevel modeling for systems engineering not only supports model evolution, it is also an intuitive way to describe systems. When describing complex circumstances with natural language, one describes basic, intermediate concepts first and then uses these concepts to describe the actual matter. Transferring this approach to model-based systems engineering corresponds to using multilevel modeling. This chapter develops a process, which can be used to enable model-based communication between stakeholders throughout different project phases by dynamically describing systems over their complete live cycle.

This chapter is structured as follows: at first, this thesis elaborates the challenges of multilevel modeling for a model-based systems engineering processes. Then, it describes how multilevel modeling can be used in such an engineering process and what kind of further modeling concepts are necessary. As part of this, the thesis defines the concept of context-based multilevel modeling.

4.1 Multilevel Modeling in Interdisciplinary Engineering

Systems engineering usually involves several different domains working on a common system. Different domains see the system differently but might depend on each other. Their collaboration is either organized by documents or with a common system model.

Model-based systems engineering for the whole life cycle of, e.g., a spacecraft can be implemented by a model that is kept from design phases until disposal [8]. Different engineering processes are represented by domain-specific model extensions in different project phases. For example, early design studies of a spacecraft are done in concurrent engineering facilities (CEF), where all kind of engineers work together and discuss the initial design of the spacecraft [54]. The underlying process defines tasks and issues that need to be addressed to fulfill targeted goals. This kind of group work can last several weeks, depending on the complexity of the system. CEF processes at the German Aerospace Center (DLR) are supported by Virtual Satellite, a model-based tool that can provide different domain-specific editors for the shared data model. The developers' vision is to pass on this model to later project phases until disposal of the spacecraft.

This thesis aims to combine this vision with techniques enabling multilevel concept instantiation. Figure 4.1 shows how such an engineering process could look like. A generic data model exists across all system engineering phases and is extended by domain-specific contents via defined extensions in the different phases. The example in Figure 4.1 shows domain-specific extensions for thermal, mechanical and electrical engineers in the data model for Phase 0. The next Phase A model contains additional contents for software engineers. The figure's structure is similar to the concept devel-

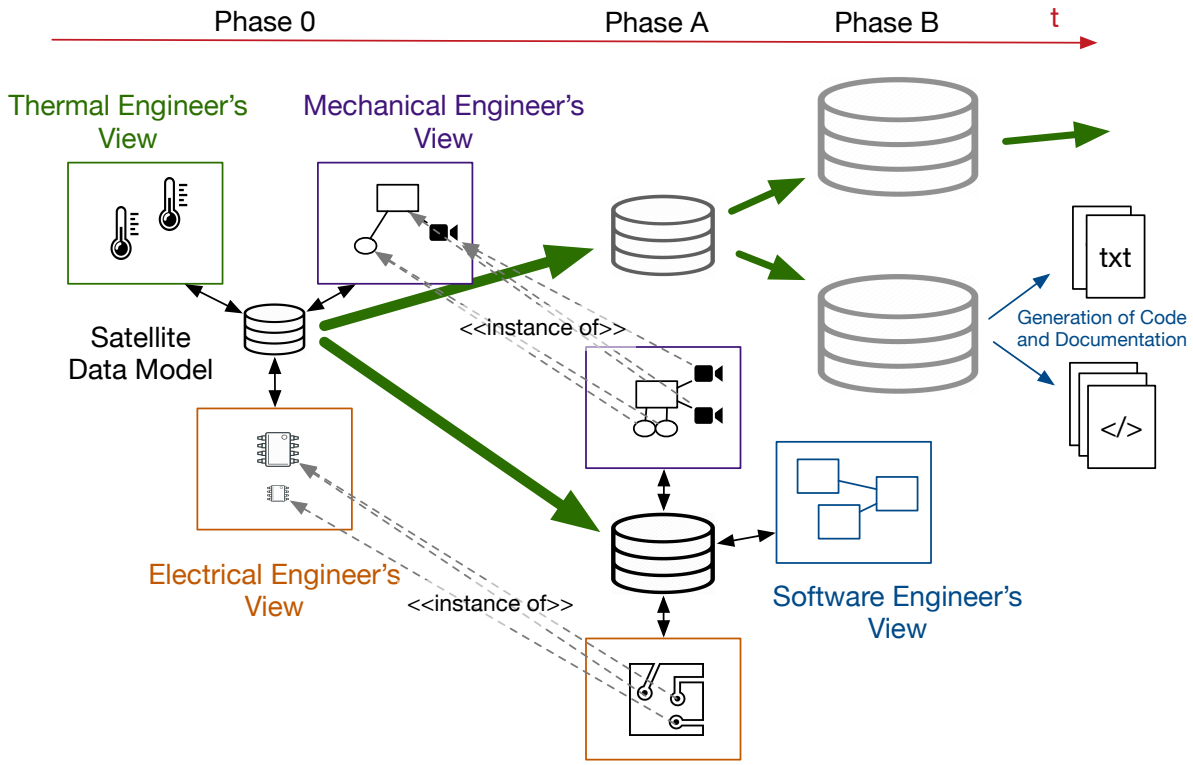

Figure 4.1: Multilevel concept instantiation in a model-based systems engineering process. Different engineers see their relevant contents with domain-specific editors. The model can be source for generation of source code and documentation.

opment, presented in Figure 1.1. Concepts in later projects or project phases instantiate concepts in an earlier phase. Multilevel modeling can help to better map this logical coherence in the data model and, thus, be the answer to the challenges introduced in Section 2.1.3. Multilevel modeling can be a technique to enable the transformation from abstract to more concrete in systems engineering. Computer chips developed in one phase are likely to be instantiated in a more complex, electrical design later; sensor concepts in early phases might be instantiated by concrete cameras and then mounted to different locations on the spacecraft. Software components might be used on different onboard computers. Finally, the data model might not only be used for communication between stakeholders, but also for generation of software and documentation. Information about the electrical design and used software components can be used to generate drivers from the data model [18].

Literature presented in Section 2.2 explains how multilevel modeling can be used to dynamically create and instantiate types. These concepts can be used to fulfill the dynamic type instantiation, demanded in Requirement REQ.1. While there exist instantiation constraints and, thus, Requirement REQ.2 can also be fulfilled, their stiff nature restricts necessary engineering dynamic and variability in project complexity. A model-based systems engineering process with multilevel modeling needs some kind of constraints that allow to adopt to changing requirements and different complex projects. Furthermore, these constraints should be understandable for non-experts in modeling.

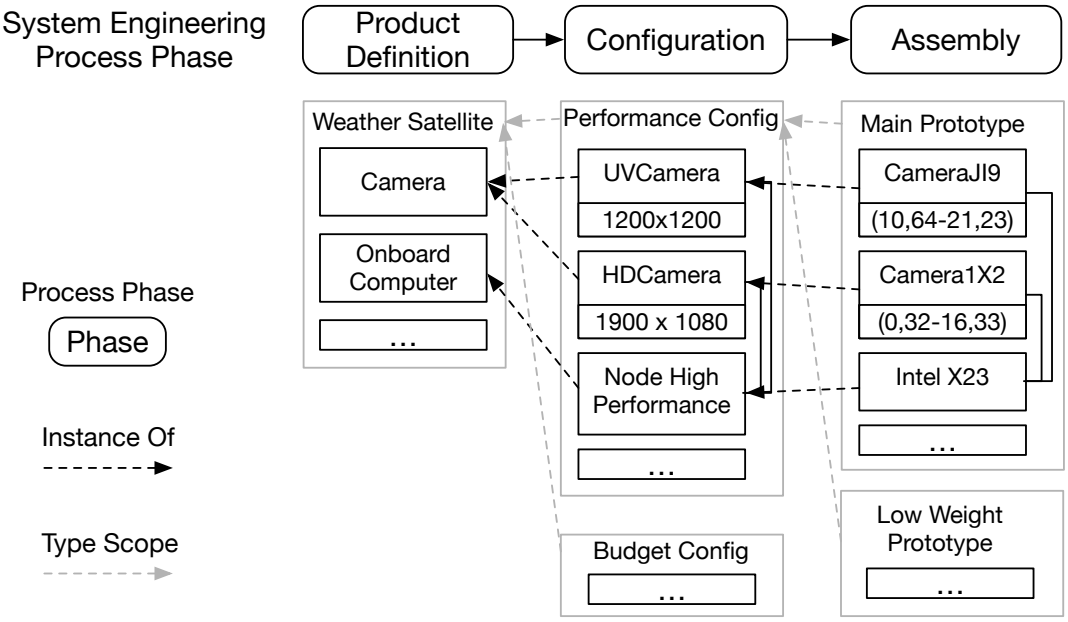

Figure 4.2: Multilevel model-based development process with three phases. The first phase is the conceptual definition of system elements. Then, in the configuration phase, the product *Weather Satellite* gets two configurations. As next step, the different configurations are assembled as prototype.

Dynamic type creation in multilevel modeling brings a high degree of flexibility. While this is one of its key benefits, there needs to exists clear control mechanism to use it in systems engineering. As Requirement REQ.7 states, the multilevel modeling environment needs to have a constraint mechanism to restrict dynamic extension of types. While current multilevel modeling literature shows ways to constraint type and field instantiation, these methods do not consider when and how fields can be added to clabjects. With the possibility to instantiate clabjects, however, this corresponds to extending types. As a result, multilevel modeling for systems engineering needs a constraint mechanism that can specify when fields can be added to clabjects dynamically.

Summarizing, to make multilevel modeling applicable for systems engineering, there needs to exist a flexible constraint mechanism that is understandable for domain experts and that restricts dynamic type extensions. The following section describes how such a mechanism can look like.

4.2 Multilevel Model-Based Engineering Process

In systems engineering, ideas of a system develop from a generic concept to concrete applications. As Section 1.1 highlights, depending on the complexity of an application of a system, there can be different numbers of development steps necessary for this transformation. Furthermore, one concept can have different concrete implementations. Multilevel modeling is a technique that can map such a structure. System elements can be defined in one development phase and then instantiated later.

Figure 4.2 shows how such a process could look like. A first concept of a weather satellite specifies that the system contains a camera and an onboard computer. After this first phase, the system model gets more detailed. A performance configuration of such a weather satellite could have two

cameras and a high performance node. In parallel, other teams could work on a budget version of a satellite. In each configuration, the elements defined before are instantiated and can be configured. Tools could already do consistency checks, validation and simulations on this level. After all system elements are configured, configurations can be tested with prototypes. Prototypes instantiate elements configured before and can specify assembly-related parameters, such as serial ids or the exact position of the element on the prototype. Prototypes might, e.g., vary in element manufacturers or physical properties of the elements. The example in Figure 4.2 shows how multilevel modeling can be used to map the relation between elements of different project phases. This way, models can be kept for the whole life cycle of a system. Each project step instantiates concepts developed before and adds more details. The type-instance relation can also be used to trace changes in the model, while the different abstraction layers provide separation of concerns [39]. Higher level models can be used as documentation and for communication with, e.g., managers. Different domain experts can work on the configuration of the elements on middle levels, while hardware developers can later work on the assembly of the prototypes, without the need to understand the meaning of the different system components.

4.3 Context-Based Model Manipulation Constraints

Using shared models for systems engineering requires strict rules when and how a model can be manipulated. This is especially true if the same model is used for the complete life cycle of a system and spans different abstraction levels. As argued in Section 4.1, existing constraint mechanisms for multilevel modeling are not appropriate for model-based systems engineering. These mechanisms, such as Potency, are hard to predict and for non-experts in modeling difficult to understand.

Figure 4.2 shows that different properties of the model elements are set at different project phases. The serial id, for example, is not available before the assembly phase starts. Such a mapping of element properties to project or system life cycle phases is easy to understand for domain experts and predictable. In fact, editing constraints for such a system engineering model are always based on the surrounding development process. Asking a domain expert how often an element property can be instantiated would not work, whereas the question when, in the development process, it can be edited is more likely to be successful. Editing constraints should not only be based on the system model but have to also consider the model's context.

To be able to create context-based constraints it is necessary to formally specify the context. Such *context models* describe how domain elements develop in a system model of the given application context. In combination with multilevel modeling, context models can describe from which abstraction levels such elements can be observed. Adding anticipated development levels of domain elements allow to specify how elements are expected to be used. As the goal for systems engineering is to constrain the usage of the model's system elements in the development process, it makes sense to describe the development process as context model.

Figure 4.3 shows how a *process-based constraint mechanism* can look like. Per domain in the data model there is a defined *context* that can be used to specify when element properties are editable. Each instance of the element is then in one of the phases, which are defined in the context. In a process, as in Figure 4.2 for camera elements, one could define that the property resolution is editable in all phases until configuration. To prevent distraction, one could define that position and serial id of the elements are visible in the assembly phase only.

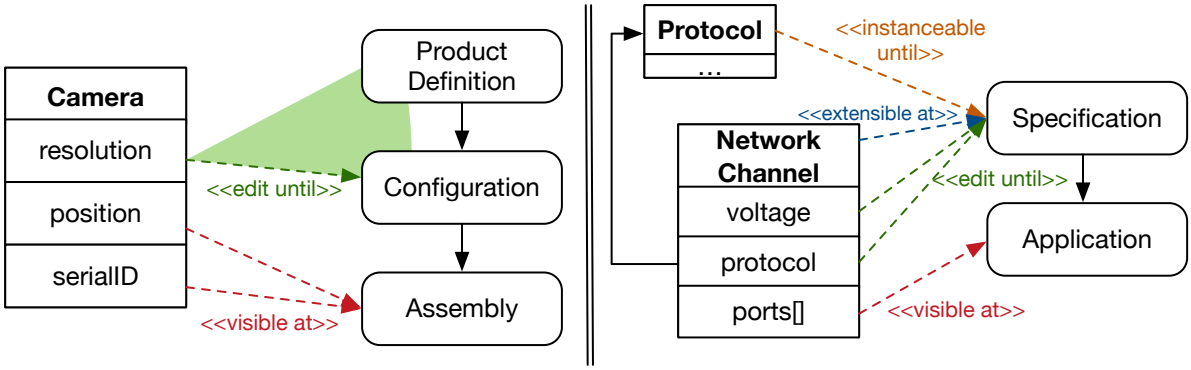

Figure 4.3: Process-based model manipulation constraints. Each domain extension can specify a context, which can then be used to evaluate when an element is visible, can be edited, is extensible or when it can be insanitated.

Besides these editing constraints, this mechanism can also specify when elements can be instantiated. On the right side of Figure 4.3, an extension defines elements to model a network. Its context specifies that channel types can be created in the specification phase and then instantiated in the application phase. As part of the specification, domain experts can create a protocol that defines how data are transferred. Later, in the application phase, channel instances of the previously defined network technology can be connected to ports. Because the protocol cannot be changed in this application phase anymore, the protocol definition can not be instantiated. Such a constraint can be modeled by, e.g., specifying that an element can be instantiated until a specific phase.

To fulfill Requirement REQ.7, there needs to be a mechanism to restrict the extension of types when using multilevel modeling for model-based systems engineering. This can also be done process-based. As mentioned before, Figure 4.3 shows an example where domain experts can specify a network technology and later use it. The constraint mechanism allows that at the specification phase *Network Channel* elements can be extended but not in the application phase. This way, the modeling environment provides all flexibility of multilevel modeling but in a controlled way.

Context models need to be separated from application models. They define how application model elements can be interpreted. As clabjects show their class or object facet depending on the context, application model elements can have different facets depending on the context model.

Different domain elements might use the same context definition and in one application model can be domain elements with several different contexts. Furthermore, different context definitions can be used to adopt to different complex projects. A project can be based on a model of the first two phases and their abstraction levels but use a more complex engineering process. As, e.g., shown in Figure 4.4, the context could then be redefined to have an additional integration step. Context-based constraints can be based on phases that are known when creating the constraints only but range-keywords, such as until, after or between, can be used to design them in a farsighted way. When specifying that a field value can be changed, e.g., between two phases then this field can also be edited in phases that are added to this range. In Figure 4.4, the position of elements is visible in the integration phase because the constraint specifies that it is visible *after* the configuration phase. This is true for the integration phase. If one wants to exclude new phases, explicit addressing of

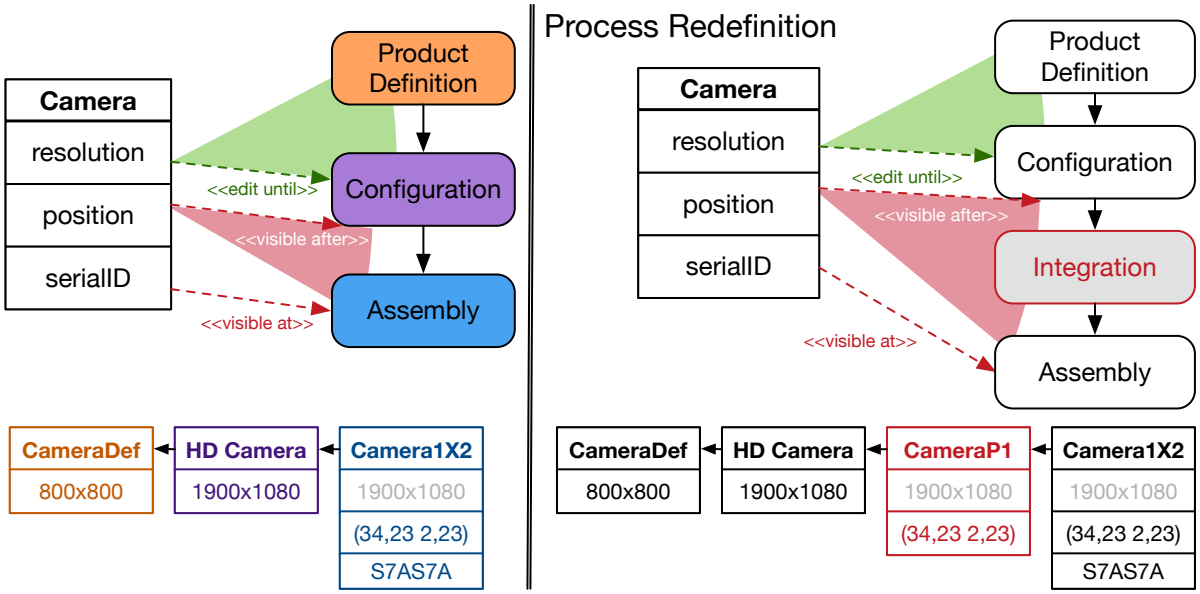

Figure 4.4: Process-based element instantiation and context redefinition. Depending on the process-based constraints, element properties are mapped to the instances. Range constraints can be used to consider process redefinition.

Table 4.1: Specifiers for context-based model manipulation constraints.

Element	Type	Phase Specifier
Clabjects	instanceable	always
	extensible	at *Phase A*
Fields	editable	after *Phase A*
	visible	until *Phase A*
		between *Phase A* and *Phase B*

phases, such as *at Assembly*, is possible.

To properly create such context-based constraints, a textual language can be used. Such a language should provide functionality to constrain edition, visibility, instantiation and extensibility. Constraint definition for clabjects and fields can be based on the structure as specified in the Table 4.1. The *extensible* keyword allows to add new field definitions to clabjects. By default this is not possible, extensibility of clabjects has to be explicitly allowed.

Process definitions can also be more complex than a linear list of phases. Alternative paths to e.g. previous phases can be used model iterative processes. Instantiation then follows the main path by default but allows, e.g., edits from alternative paths and to manually switch to alternative paths.

Process definition not only allows to specify constraints for model manipulation but can also be used to create phase-specific views on the model. Domain experts could then select their domain and current process phase to apply a filter to only see relevant elements.

4.4 Terms for Context-Sensitive Multilevel Modeling

To improve communication about context-sensitive modeling and to clarify the wording in this thesis, this section defines some terms.

Context-sensitive multilevel modeling This kind of modeling describes a methodology where multilevel modeling is combined with a modeled context that describes how multilevel elements can be instantiated and to define model manipulation constraints.

Context The context describes how elements can be instantiated in a multilevel environment and how they have to be interpreted. A context phase represents a model level.

Context-based model manipulation constraints These kind of constraints uses the multilevel context to specify when elements can be instantiated, edited and extended.

Model level labels Context-sensitive multilevel modeling removes the level numbering and uses multilevel context phases to label the levels for the different domains. For an engineering process as context description, levels are labeled with process phases. The same level can be labeled with different process phases for elements of different domains. A global schema for labeling of levels, which can contain an arbitrary number of technologies and concepts, is too rigid. Depending on the project or technology, additional instances of a concept might be necessary. Furthermore, as Kühne argues, in an order-aligned scheme it is possible to shift a local classification ensemble, such as elements of one domain, up and down in a global level hierarchy [55]. The context phases order-alignment of domain concepts corresponds to Kühne's *Local Total Order-Alignment*.

For model-based systems engineering, it makes sense to model the engineering process as multilevel context. Element instances can then be mapped to process phases. As one of the goals of this thesis is to make multilevel modeling applicable for systems engineering, the remainder of this thesis focuses on process models as context.

Process Multilevel modeling *context* for model-based systems engineering. The process should have a default path but can have alternative engineering steps.

Process-based manipulation constraints Context-sensitive constraints based on a process as context description.

Phase A *phase* is a part of the process description that describes the instances of an element.

5 Implications on Environment Implementations

To implement multilevel modeling in common modeling envionrments, an arbitrary number of model levels has to be mapped to the classical two-level paradigm. According to Atkinson and Kühne it makes sense to use multilevel modeling even if the solution technology does not support the required number of modeling levels [40]. They suggest to create clean models, with the number of levels needed, and to transform them to a supported form later. This chapter explains how context-sensitive multilevel modeling can be mapped to two-level environments.

One of the key success factors for model-based projects is tool support and required training for working with models [1]. However, to enable multilevel modeling, generic concepts, such as the clabject, need to be in the linguistic metamodel. As a result, the infrastructure derived from the linguistic metamodel is not domain-specific and, thus, requires more learning effort. While modeling environments can be customized dynamically [49], a complete redefinition of modeling languages is not possible. Furthermore, implementing custom validators, simulation environments and generators based on such a model, requires to query elements by name and cast values to their expected type. Because this is an error-prone process, model-based environments, which use multilevel modeling, should generate some kind of supporting infrastructure. Only then are domain specific editors and artifact generation possible and Requirements REQ.5 and REQ.6 can be fulfilled.

This chapter is structured as follows: the first part highlights the importance of domain-specific infrastructure. Section 5.2 introduces the concept of domain metamodels, which allow to specify domain types, and allow to create domain-specific infrastructure. Based on domain metamodels the next section explains how model elements' representation can be customized. The last section explains how domain-specific infrastructure can be used for editors and artifact generation.

5.1 Domain-Specific Infrastructure

One of multilevel modeling's greatest benefits is that types can be created without the need to update the metamodel. In the classical two-level paradigm, the linguistic metamodel is used to define domain types and to generate their infrastructure. With multilevel modeling, however, the utilization of this linguistic metamodel to an infrastructure definition, which enables ontological types, leaves a gap that needs to be filled. While domain types in multilevel modeling are defined in an ontological metamodel, their infrastructure is absent.

Model-based systems engineering lives from domain-specific extension implementations, such as project-specific validators, editors or artifact generators [28]. While this is still possible with dynamically created types, the implementation based on such generic infrastructure takes much more effort. Figure 5.1 shows one of the reasons why: with generic infrastructure, accessing a model element requires to query elements by their name and to cast values to their expected type. This is not

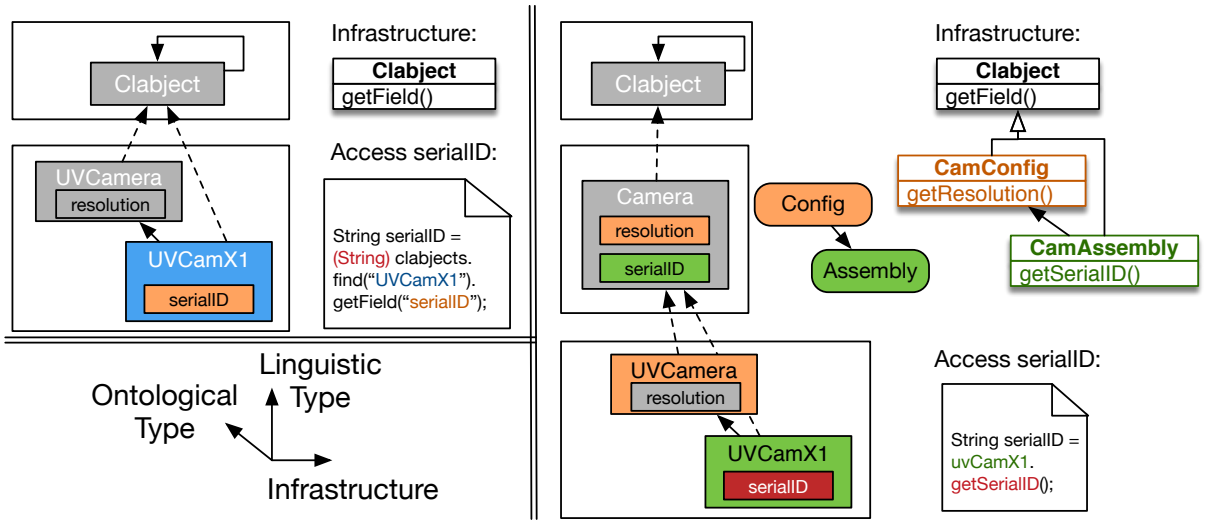

Figure 5.1: Mechanism for domain-specific infrastructure. The left example shows classical multilevel modeling. The right part of the figure shows multilevel modeling with domain-specific infrastructure. For both methods, the UML-note shape contains code examples.

only costlier but also more error-prone than accessing the value by a, from a linguistic metamodel, generated getter method.

As the right part of Figure 5.1 suggests, adding an additional extension layer, which supports the concepts of multilevel modeling, can help to solve this problem. A definition of concepts and their properties in such an extension mechanism enables to generate infrastructure for these concepts while also supporting multilevel modeling. With process-based multilevel modeling, as described in Section 4.3, such a mechanism can also generate dedicated infrastructure for each process phase. Infrastructure then supports dynamic properties as a generic infrastructure does but has additional methods for properties that are defined in the extension mechanism. This way, concepts and properties which are known when setting up the extension can be supported by a generated infrastructure while concepts that are only known at runtime can be handled with ontological typing.

5.2 Domain Metamodel and Context Scope

Infrastructure generation for domain elements can be, as presented in Section 2.1.3, based on the definition of a system element's domain type. Domain types can be grouped to a set of elements of one domain. As context models describe how domain elements develop in the system model, this information should be part of the domain type definition. Groups of domain types can be used as scope for a context and the context model can be used to generate phase-specific infrastructure. In this thesis, the combination of a set of domain types and a corresponding context are called *domain metamodel*. Domain metamodels can contain context-based constraints.

Model-metamodel co-evolution requires to migrate the instance model every time the metamodel is touched [28], whereas the definition of new types in such a layer must not require to migrate the system model. As presented in Figure 5.2, this can be archived by a static metamodel that is basis for multilevel modeling concepts, such as clabject, and concepts for an extension mechanism of domain types. This way, instances of new domain extension are linguistic instances of the static metamodel

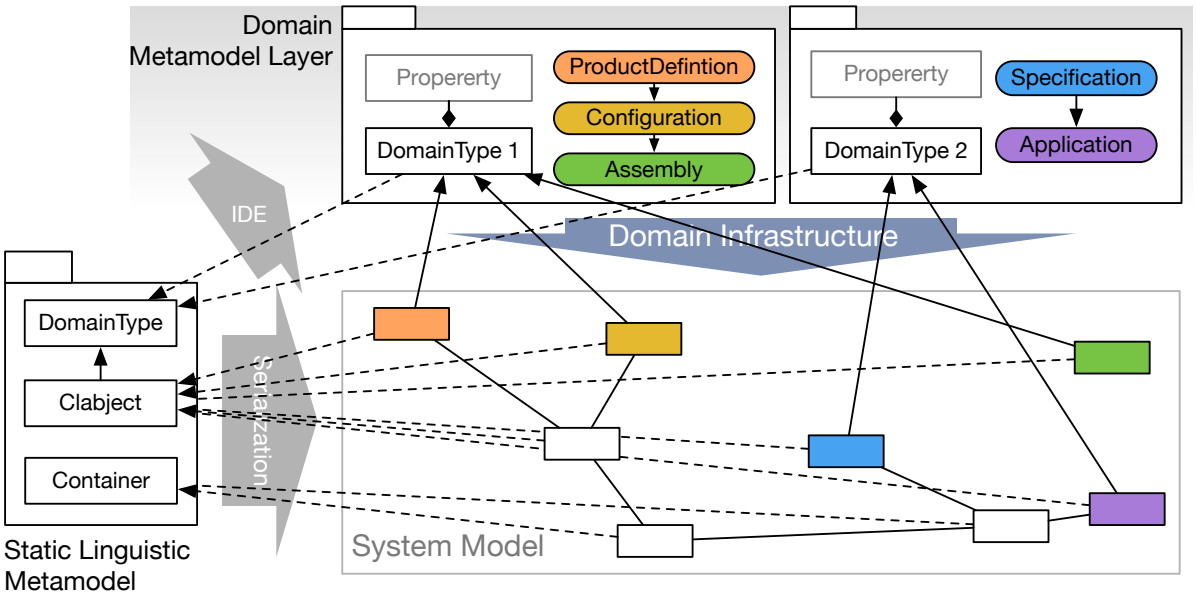

Figure 5.2: Structure of an environment with an additional metamodeling layer. Infrastructure is generated from the static metamodel as well as from the domain extension layer.

and can be used in the system model in the same way as older domain types. Furthermore, as their nature of clabjects, these instances of the domain extensions can be instantiated again in the data model. Because the static metamodel defines concepts for the domain metamodel, the environment can provide a development environment for these extensions. This way, it can support creating multilevel modeling contexts and context-based constraints by providing context editors.

As Figure 5.2 shows, system model elements can have three types. As the static metamodel is responsible for serialization of elements, each model element has a linguistic type. Furthermore, as defined in the static metamodel, clabjects can have a domain type. Finally, because of their implementation as clabject, elements can have an ontological type. Domain types are modeled a separated development environment, which then generates infrastructure for these types. As Figure 5.2 presents, the system model environment is based on a combination of infrastructure from the static metamodel and the domain metamodel layer.

Figure 5.3 shows a visualization of the orthogonal classification architecture with the integration of an additional classification dimension for domain type definition. This way, multilevel modeling can have a dedicated dimension for the definition of domain types and it is possible to derive domain-specific infrastructure. The domain metamodel extension layer fills the gap that the utilization of the linguistic metamodel for multilevel modeling concepts left. Modeling in such an environment combines the benefits of the classical two-level paradigm with multilevel modeling. Domain types can be specified in a dedicated layer with infrastructure generation and these application model elements can be instantiated several times to model classification hierarchies within the application model. This way, the classical linguistic dimension is separated to a dimension for the definition of static language types, which are serialized, and a pseudo-linguistic dimension, which describes the domain infrastructure for each element. This has the benefit that new domain types

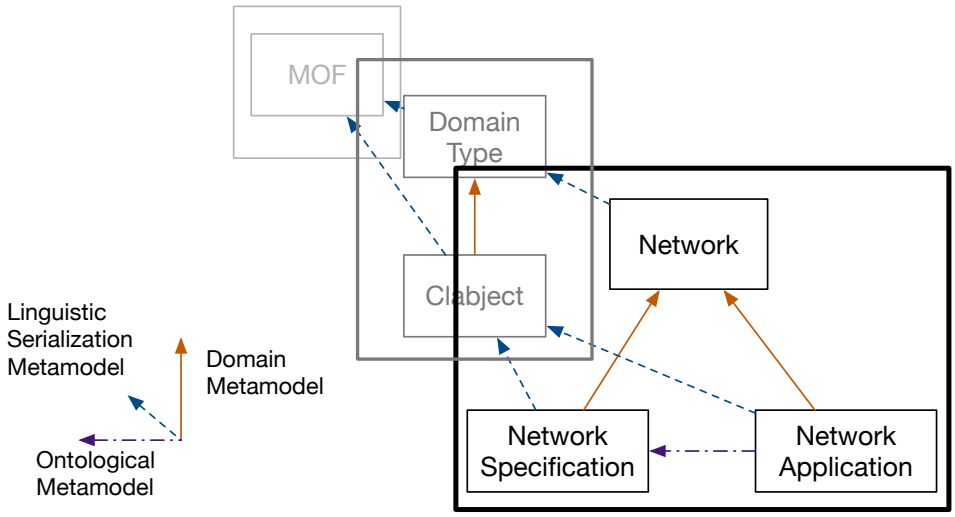

Figure 5.3: Three dimensional orthogonal classification architecture. Graphical visualization of the third classification dimension for domain concept definition integrated into the classical orthogonal classification architecture.

can be introduced without the need to migrate the instance models. Thus, the domain metamodel dimension is not a real linguistic dimension but rather something that describes how the abstract syntax can be interpreted.

The example in Figure 5.3 shows the meaning of each dimension. The serialization metamodel dimension describes the domain independent aspects of an element and is used to derive domain independent infrastructure, such as the serialization. System elements are based on concepts such as the clabject because the goal is to use multilevel modeling. The domain metamodel dimension describes the domain-specific aspects of a model element and can be used to derive domain-specific infrastructure. The domain *Network* could, e.g., model that elements can be connected to each other and provide infrastructure for accessing connected elements. Instances of this network domain element could then specify a network technology and how it communicates. Because model elements are implemented as clabject, this network technology can be instantiated and used to describe the actual connections of model elements. Thus, the ontological metamodeling dimension can be used to describe classification relations within a data model.

Process-based model manipulation constraints can be attached to elements in the domain metamodel layer as well as to clabjects in the system model. This way properties, which are added dynamically at runtime can also be restricted to specific process phases.

5.3 Dynamic Model Representation Customizations

While the domain-specific infrastructure enables to implement domain-specific editors from scratch, generic editors should also be able to visualize the system model elements in domain-customized ways. As Atkinson and Gerbig present, tools for multilevel modeling can implement domain-specific editors by dynamically specifying graphical representations for clabjects [49]. Similar to the visualization search algorithm, described in Section 2.2.5, a tool based on this three dimensional orthogonal classification architecture can use graphical element representations specified for its

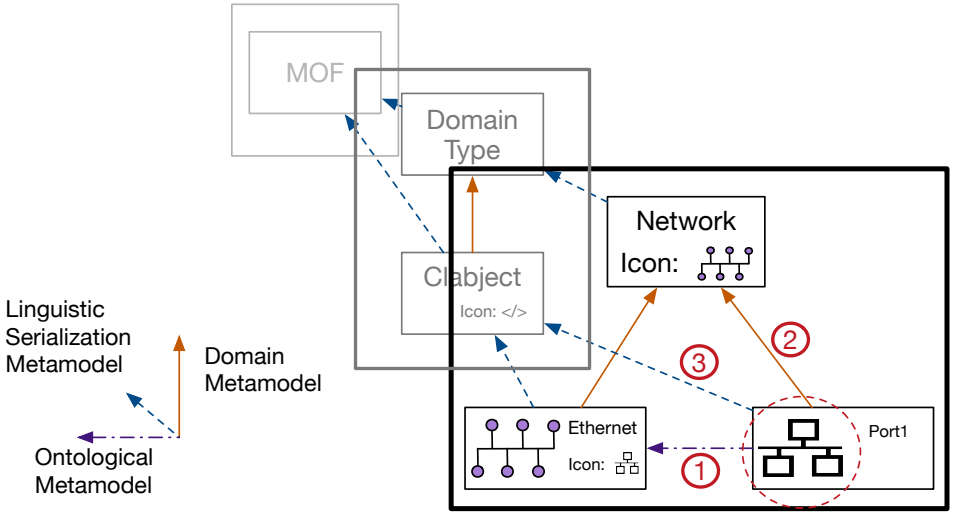

Figure 5.4: Dynamic element representation. Order of search scope for element representations.

types in the different dimensions. Figure 5.4 shows in which order the dimensions are searched. If there is a representation specified for the ontological type, then this representation is the first choice. If the ontological type does not have a representation specification its own ontological type is checked. If there is no graphical representation specified for any ontological type, then the algorithm searches in the element's domain metamodel. If there is no representation specified either, the element's linguistic type representation is used. Figure 5.4 shows an example for this algorithm. The element *Port1* uses the icon specified in its ontological type *Ethernet*. If an element does not have an ontological type representation, as for the element *Ethernet*, then the representation from the domain metamodel is used. The element *Ethernet* uses the icon specified for network elements. If there was no representation in the ontological type hierarchy and in the domain metamodel, then the element would be visualized with the representation specified in the linguistic type, here the *Clabject* element.

The specification of element representation can be as simple as the definition of icons but it can also be more complex. The environment could use a domain-specific language to specify element viewers and editors. Such a language can be used to create graphical and textual languages as well as also to design a user interface. However, to clearly separate layout and domain data, the content of such a language should be separated from the definition of the domain elements.

5.4 Domain-Specific Editors and Tool Support

Domain-specific infrastructure facilitates the development of modeling tools. Paige et al. highlight the importance of external interfaces of the modeling environment [28]. Model-based systems engineering benefits most with a high degree of automation and tool support. Furthermore, as Frank argues, multilevel modeling can be used to model different abstraction layers of projects in organizations and, thereby, connect people in different hierarchy levels [30]. Thus, it makes sense to also create separate tools for the different model levels. Higher level models, e.g., can be used to create documentation of the general concept of the project. Lower level models can be used to generate specialized source code. The instantiation from generic concepts to concrete implementations can

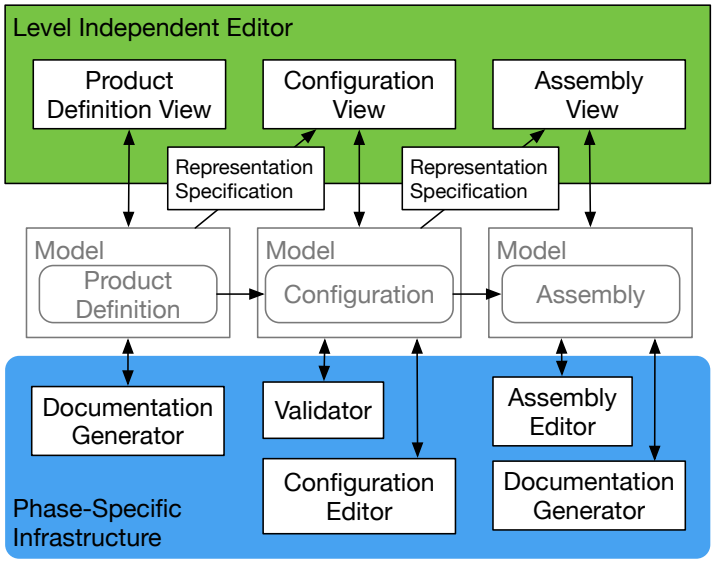

Figure 5.5: Domain-specific editors and tool environment for models with several levels.

be used to trace elements through the system or project life cycle. Multilevel modeling helps to connect different project phases and, thus, is a natural support for model-based systems engineering. Model-based tools help to make this support applicable for projects.

Figure 5.5 shows how such a tool environment could look like. The level independent editor is based on the linguistic metamodel and thereby supports all model levels. Element representation specification, as introduced in Section 5.3, can be used to customize this generic editor. Such an editor makes sense for the general model overview. However, for system development, more specialized tools make sense. Different engineering tasks on different abstraction levels and process phases suggest to create specific tools for the phases. As shown in Figure 5.5, it might make sense to create a documentation generator based on the highest modeling level to explain the general concept of the project. If elements have to be connected in the configuration phase, then a graphical editor could be helpful. Validation could check if the created configuration works.

5.4.1 Domain-Specific Editors

Phase-specific editors can be build on domain-specific infrastructure, as introduced in Section 5.1. Model implementations should provide a mechanism to access visible features. As the context model defines how system elements are expected to be used, editors should consider this description. While some element properties might not be visible in some phases, editors should also check that executed edits are valid in the given context. As a domain engineer might work only on one domain at a time and only on the domain aspects of the current process phase, editors can implement filters to hide elements of other domains and model levels.

5.4.2 Artifact Generation

Multilevel models can exist for the whole life cycle of a system and describe it from different abstraction levels. Furthermore, with a tracing from early conceptual phases till concrete implementations, the models contain information and coherencies that can be used to create documentations

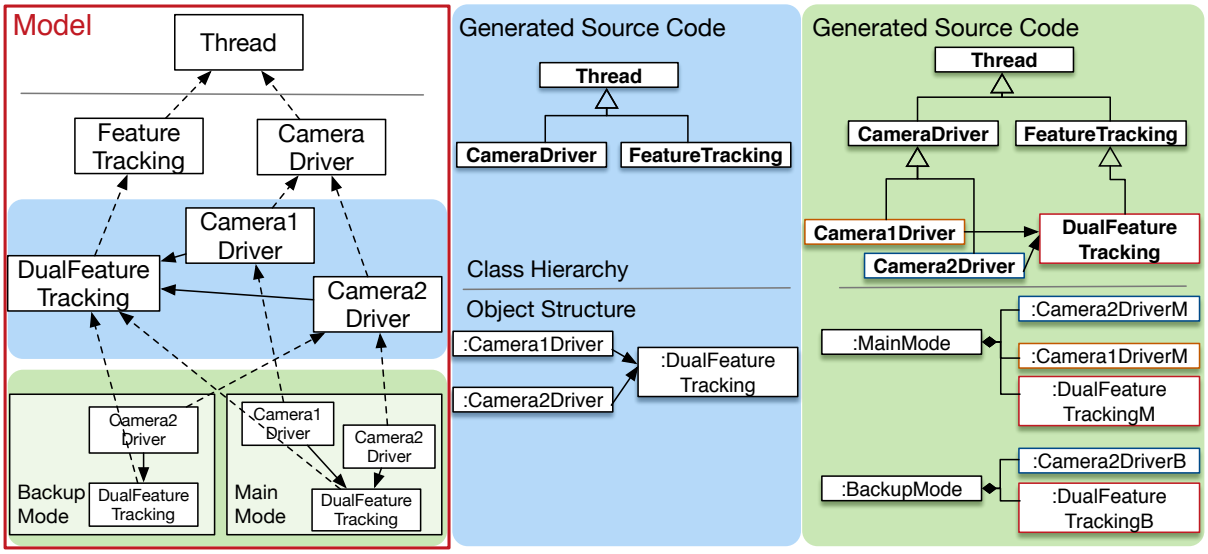

Figure 5.6: Code generation from multilevel models. The multilevel model is shown in the left picture. The middle part shows code generation from third model level, the right part shows code generation from the fourth model level.

and to generate source code. Requirements specification documents, for example, should contain system descriptions at different abstraction layers and trace elements throw these different layers [56]. Thus, multilevel models contain information that can be used as a basis for documents of, e.g., subcontracts or to keep system documents up-to-date. On the other hand, engineers can add system descriptions as requirements to models and later instantiate these described elements in implementations. This way, tools can automatically check that requirements are fulfilled and that system elements have a reason to exist.

As presented in Section 2.3, models can be used to generate not only documentation but also source code and build system files. With the project knowledge about hardware and software stored in the model, large part of the system's onboard software can be generated. Hardware technologies and interfaces can be used to generate drivers, network descriptions can be used to generate software communication code. These generators can use the generated domain-infrastructure and combine model elements with source templates to generate source code.

To map multilevel model content to source code, it has to be transformed from varying number of model levels to two source code levels. Generated source code can consist of classes, the first level, and object structures, the second level. This transformation can be done by generating required most concrete clabjects as objects and thier types, more abstract clabjects, as classes. Most concrete clabjects do not have a class facet, while values in more abstract clabjects can be added, e.g., into the constructor. This way, it is possible to generate source code from different abstraction layers. Furthermore, for code generation it is important at which abstraction level the generation was triggered because this level holds the most concrete clabjects. As a result, models at different abstraction levels can generate executable source code. Figure 5.6 shows an example, where a multilevel model describes a navigation system software. Camera drivers produce images, which are analyzed by a feature tracking software. The result of a code generation could be a class for the cam-

era driver and for the feature tracking. As both, feature tracking and the camera driver are a thread their class would extend the thread class. With a code generation from the third model level, the connection of the elements results in source code with objects of the camera driver class for both cameras and an object of the feature tracking class. Both camera objects would then send data to the feature tracking object. A subsequent project, which develops safe mode configurations would instantiate some of the elements in different configurations of the navigation software. As shown in Figure 5.6, code generation from this, here fourth, model level would then result in separate classes for both camera drivers. Each camera driver class can have an instance in the mode definitions. If, however, the different camera drivers in the model do not define additional fields, code generation could be simplified by just creating four instances of the camera driver class. Furthermore, if a project's goal is not to generate executable software, code generation could just generate classes. Such a generation would make sense from the first two model levels in Figure 5.6.

6 Multilevel Model-Based Tool for Space System Engineering

Multilevel modeling increases modeling flexibility compared to two-level approaches. Space systems engineering requires strict rules because due to its isolated environment, potential failures are difficult to compensate. Thus, an application of multilevel modeling in this domain has high demands on constraint mechanisms. This chapter shows how context-sensitive multilevel modeling can be implemented in such a critical engineering domain. The implementation is integrated into Virtual Satellite, a model-based tool for space system engineering.

To demonstrate its generic applicability, the multilevel modeling environment is implemented in a Virtual Satellite independent way. Other model-based tools for systems engineering can implement its interface and use the multilevel modeling capabilities. Furthermore, to show how the multilevel modeling environment can be used in tools, a separate tool environment with a generic metamodel and an extension mechanism for systems engineering implements the multilevel modeling interface. This generic systems engineering tool environment is suitable as basis for new multilevel model-based systems engineering tools.

Figure 6.1 shows the implementation layers of the environment, its integration into Virtual Satellite and the generic tool environment. Both, Virtual Satellite's core and the generic multilevel modeling environment are implemented on top of Ecore.[1] The multilevel capabilities are implemented within the model environment's linguistic infrastructure and can be used in all model editors, such as the Virtual Satellite editor or even the generic, generated EMF editor.

This chapter is structured as follows: The first part describes how the generic environment for context-sensitive multilevel modeling is implemented. Thereby it presents the editors for domain metamodels and how context-based constraints are resolved. Based on this environment, the next section describes the integration into Virtual Satellite. The last section describes how multilevel model element's representation can be customized dynamically.

6.1 Context-Sensitive Multilevel Modeling Environment

The multilevel modeling environment is based on a linguistic metamodel, that defines basic concepts for multilevel modeling, such as *Clabject*, *Field* and *ClabjectContainer*. Besides these concepts it also defines concept for context-sensitive modeling, such as *Context* and *Context Phase*. Instead of potency and durability *Clabjects* and their *Features* contain *ContextConstraints* to restrict their instantiation and edits. To provide domain-specific infrastructure, the metamodel contains concepts for the domain metamodel, such as the *DomainType*.

As shown in Figure 6.1, on top of this multilevel metamodel, the environment contains multilevel services that help to instantiate multilevel elements and to maintain their context. As discussed

[1] Part of the Eclipse Modeling Framework, webpage: https://wiki.eclipse.org/EMF

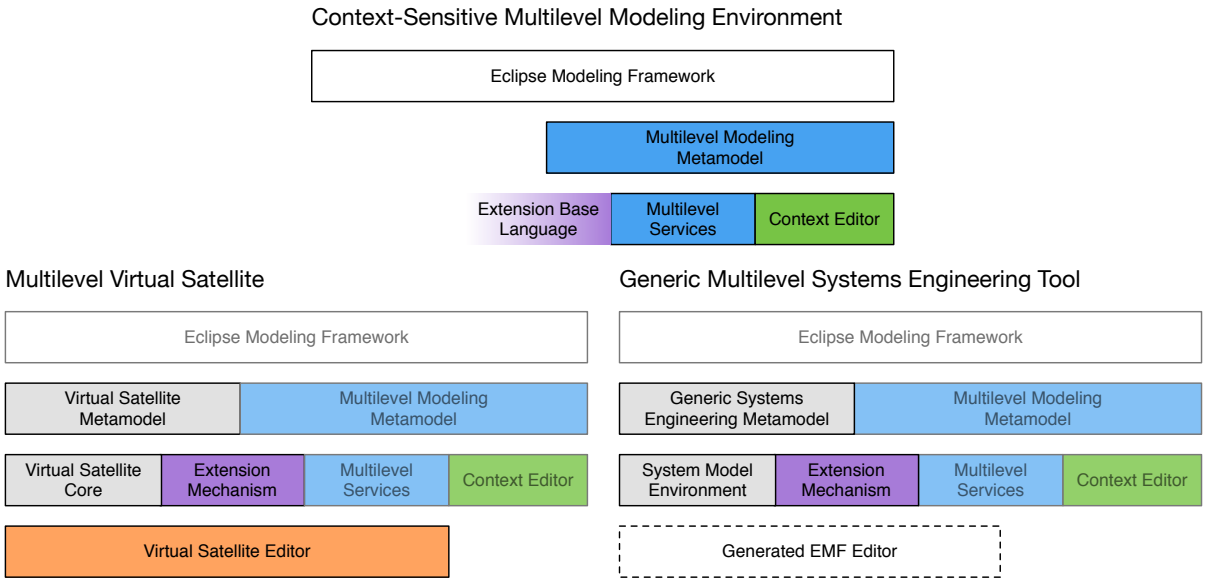

Figure 6.1: Implementation layers of the mutlilevel modeling environment and its integration into Virtual Satellite and the generic systems engineering tool.

in Section 2.2.6, the implementation of this instantiation operation of elements is based on the prototype pattern and its first operation is to copy the type element to the instance level. After that, the type-reference is set and default values are instantiated. Such an instantiation operation handles child elements recursively. The context service maintains a clabject's context once it is created or instantiated. It automatically updates the context and resolves context-based constraints.

6.1.1 Domain Metamodel Development Environment

As shown in Figure 6.2, creation of domain elements and their context definition can be done in an integrated development environment. With the definition of the domain metamodel in the linguistic metamodel, the environment can provide editors for domain elements and their context. While the generic multilevel modeling environment does not provide an extension mechanism to create domain types, it contains a base language that can be used to specify context-based constraints. This base language has grammar rules to create constraints, such has `after Configuration` or `at Assembly`. Tool-specific domain extension mechanisms can extend this base language to use the context-based constraints for domain elements and their properties.

Furthermore, the development environment contains a graphical editor for creation of the context model. The graphical editor is based on the graphical editing framework[2] and its resulting context model can be referenced in concrete domain extension mechanism to specify context-based constraints. This way, different domains can use the same context definition. As the environment targets systems engineering, the graphical context editor provides editing capabilities to model the development's underlying engineering process. In the graphical editor, domain experts can create process phases and connect them. Besides a main development path the editor supports alternative paths and enables iterative processes.

[2] Graphical Editing Framework; webpage: http://www.eclipse.org/gef/

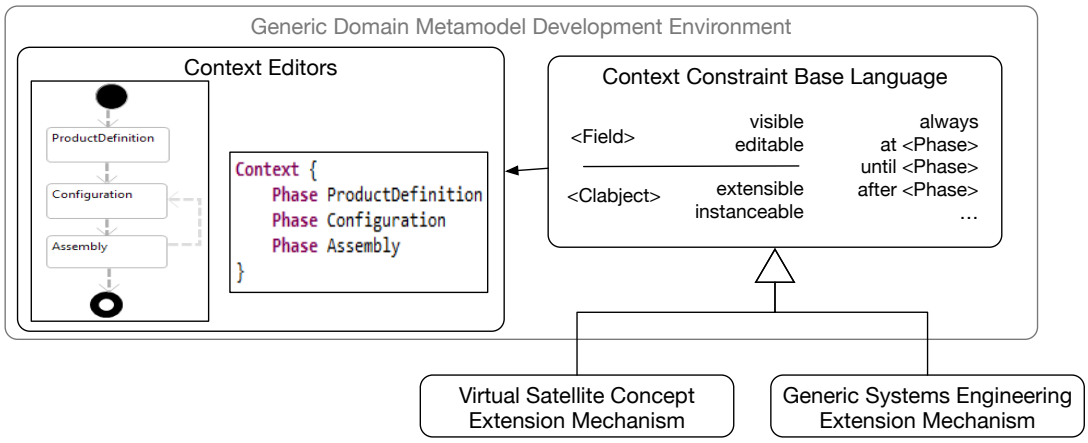

Figure 6.2: Domain metamodel development environment. Context editors can be used to create the context model. The constraint base language specifies how constraints can use the process phases.

Besides this graphical editor for context modeling, the environment contains a simple textual editor that allows to specify a context within the textual definition of domain elements. This inline context editor allows to specify a linear order of context phases.

There are two options to redefine the context model of a project: Firstly, *explicit* redefinition is done with the context editor. Such an explicit edit of the context allows to completely change all phases that do not yet have an implementation in the corresponding system model. Furthermore, it is possible to add new phases to the context. Deleting an existing phase is not allowed as it might be referenced in a constraint. The second option to update the context is *implicit*. If an additional instantiation level is required, the editor can just clone the previous phase.

The base language for context-based constraint definition is not only restricted to the extension mechanism but can also be used to dynamically add constraints to clabjects and their fields at runtime. This way it is possible to constrain fields, which are added to extensible clabjects.

6.1.2 Multilevel Service and Constraint Evaluation

Part of the multilevel model infrastructure is a context service that maintains the context descriptions and evaluates the context-based constraints. As shown in Figure 6.3, to support different context descriptions, the actual accessing of the context model is outsourced to a dedicated class. This way, adding a new kind of context description, such as process models, can be done by implementing a context editor and a *ContextAccessor* class. To support model-based systems engineering, the environment provides two different kinds of context modeling. The first, simple one is a textual definition of process phases. Order of the phases is defined by their definition and this simple textual context description can be done inline, within the extension mechanism. The second, more sophisticated, type of context description uses a graphical editor.

Context descriptions can have any format as long as there exists a context accessor that can resolve context-based constraints and increase the level. With the simple process context, shown in Figure 6.3, the accessor compares the index of the phases in the definition-list to check if phases are before or after the current phase. Process contexts that are modeled in the graphical editor require to iterate through the context model to check whether phases are reachable after the current

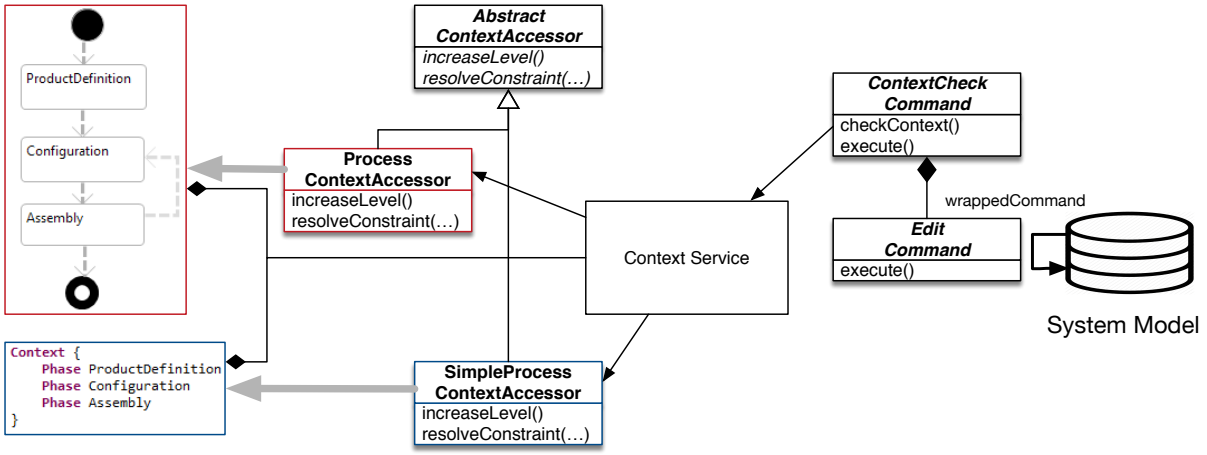

Figure 6.3: Implementation of context-based constraints and their infrastructure. Different kinds of context description can be used if there exists a corresponding context editor and an accessor class.

phase. The context accessor also handles cycles in the process definition to support iterative processes. The graphical process definition uses a main development path to increase the modeling level. However, the resolving of constraints considers alternative engineering paths and users can switch to phases on the alternative phase manually.

As databases manage edits with transactions, models based on Ecore allow edits only within special commands [57]. As shown in Figure 6.3, the multilevel model environment customizes this command base class to check the context before executing the edit. This way, edits are only possible if no context-based constraints are violated. Furthermore, such *ContextCheckCommands* ensure that a clabject's modeling context is initialized and updated.

With an integration of the context checks into the model infrastructure, model editors automatically support context-specific editing constraints. Even the automatically generated EMF model editor rejects changes within an invalid context because it uses the *EMFEditCommands*, which the multilevel environment modifies. Visibility constraints, however, have to be implemented by the editor.

6.2 Virtual Satellite Integration

Virtual Satellite is a model-based tool for space systems engineering. As presented in Figure 6.1, the Virtual Satellite editor is based on Virtual Satellite's core and its system engineering model that, e.g., supports resolving of engineering units and mathematical expressions. The multilevel implementation of Virtual Satellite has a customized metamodel that extends the metamodel of the context-sensitive multilevel modeling environment. To be able to specify context-based constraints in its extension mechanism, Virtual Satellite's concept language extends the constraint base language.

Figure 6.4 shows the extension mechanism of Virtual Satellite and the generic systems engineering tool. As the extension mechanism is tool-specific, both tools have a custom syntax to define domain types and their properties. As both extension mechanisms are based on the context constraint base language, their notation for context import and constraint definition is the same. The import

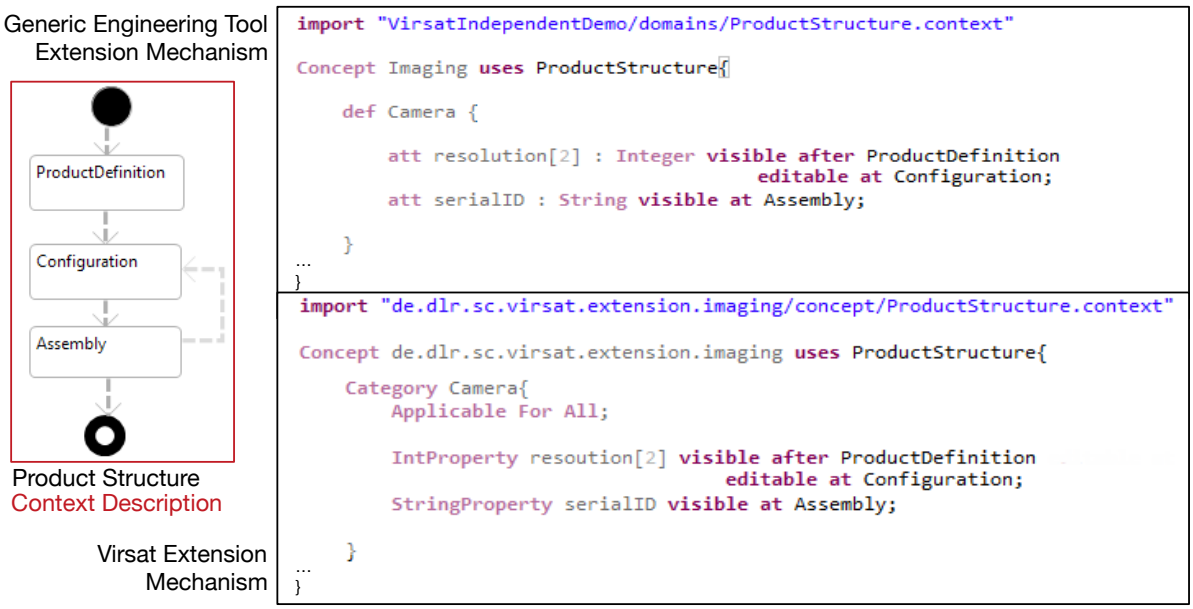

Figure 6.4: Domain metamodel development environments. Different tool-specific extension mechanisms can use the context base language to use the grammar rules for context-based constraint definition. Both extension mechanisms import the context, which editor's view is shown on the left.

allows to specify a context model file and the `uses` statement makes the context phases available for constraints. In the example, both domain extension mechanisms specify a camera domain type. This camera has a resolution property, which is visible after the *ProductDefinition* phase but can be edited only in the *Configuration* phase. The serial id is only visible in the *Assembly* phase.

In the Virtual Satellite version with context-sensitive multilevel modeling, the editor is based on the multilevel services of the multilevel modeling environment. This way, it can customize editing capabilities context-based as the editors can use the context service to query all visible elements or to check if a specific model element or property is visible. Figure 6.5 shows the editor for the camera element in the three phases as defined in Figure 6.4. The left editor section represents the camera element in the *ProductDefinition* phase. In this phase, the editor only shows the Virtual Satellite-mandatory fields, such as name and an automatically generated unique identifier (minimized in the figure). The middle editor section represents the camera element in the *Configuration* phase. This configuration element, which is an instance of the camera definition on the left, allows to modify the resolution of the camera. The next instance, shown by the editor section on the right, does not allow to edit the resolution anymore. However, as defined in the extension in Figure 6.4, the resolution is still visible. This *Assembly* representation of the camera element, furthermore, shows the serial id and allows its modification. This way, the Virtual Satellite editor shows different element representations depending on the context-phase, in which the multilevel element instance is.

Besides context-specific element editing capabilities, the editor supports element instantiation. As mentioned before, the generic multilevel environment provides an instantiation service that recursively instantiates selected clabjects and its children. Besides this service, it also contains an

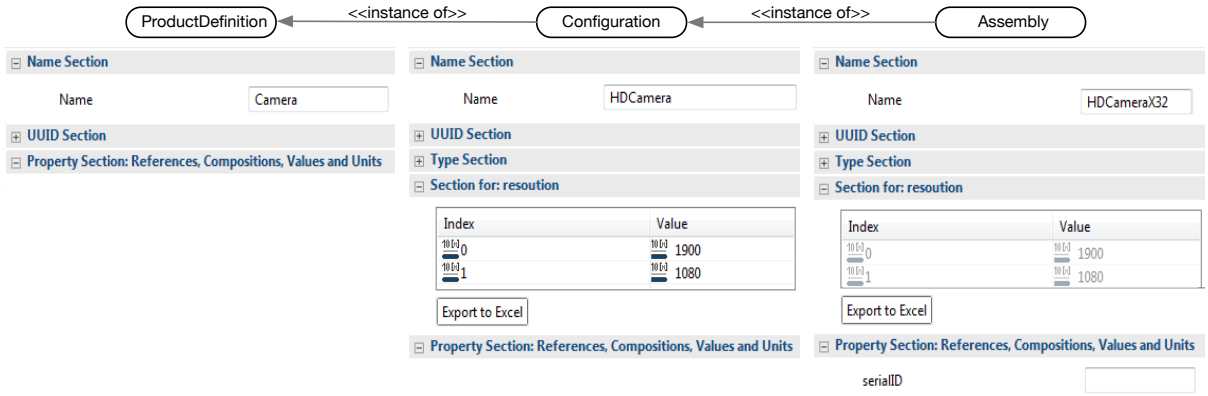

Figure 6.5: Context-based Virtual Satellite editor. The editor shows only properties that are relevant in the current project phase.

implementation of a dynamic context menu that shows clabjects on higher level that are instanceable in the given context. Scope of this context menu for element instantiation can be defined by selecting other clabject containers. Tool users could, e.g., create a container for network technologies. In the system model they could then extend the scope of this instantiation-operation to create instances of the modeled network technologies. As shown in Figure 6.6, the Virtual Satellite editor adds this context menu and additionally provides an option to automatically instantiate all elements within a selected container. This mechanism can be used, for example, to create different prototypes of a satellite configuration. Each prototype then contains all configured elements and can be customized with assembly properties, such as the serial id.

6.3 Dynamic Representation of Model Elements

The multilevel modeling environment contains a component for dynamic element representation customization. Representations can define how elements are presented in editors and viewers. A representation specification can be attached to the multilevel model elements and is then visualized by editors of the elements' instances. This proof-of-concept implementation of this *representation service* in the multilevel modeling environment provides means for specifying icons. Concrete tool environments can extend this basic implementation to also specify concrete syntax of elements in different kinds of domain specific language, such as diagrams or textual languages.

To hide complexity of the multilevel modeling infrastructure, the representation service has an interface to query the representation of elements. The service automatically searches for the most appropriate representation available in the multilevel model by considering the different type dimensions of the model. Figure 6.6 shows how this representation service is used in the Virtual Satellite editor. The modeled elements in the interface type collection, *PowerConnector* and *CanBus*, which are elements of a generic interface type have a dynamic representation specification. As a result, in the context menu for element instantiation, the editor shows the specified icons of the newly modeled types instead of the icon of generic interface types. The interface type's representation is determined in the infrastructure, generated from the domain model. The icon of the *PowerConnector*, however, is defined at editor runtime, its specification is dynamic.

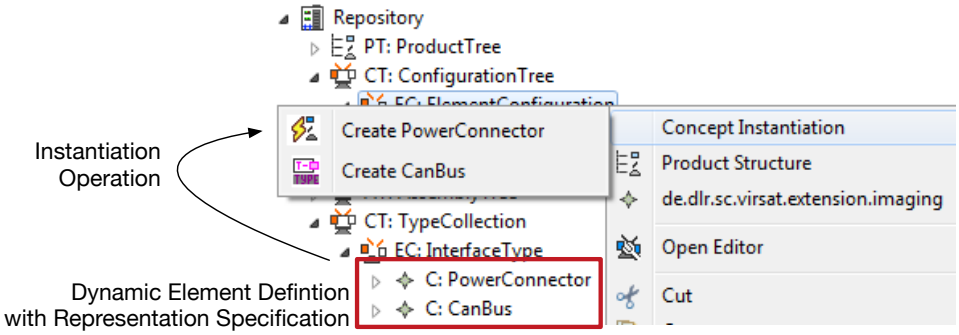

Figure 6.6: Context menu for element instantiation with dynamic element representation.

Such an element representation specification can go further than defining icons of elements. The representation service can, e.g., be extended with a DSL for tool-specific customizations of the user interface and its editors. The language could define how properties of the *PowerConnector* in Figure 6.6 are rendered on the next level and what kind of editors are available. While such a language is out of scope of this thesis, the representation service's metamodel can be used as base for such a DSL. With an integration of this basic representation specification into the multilevel modeling environment, more sophisticated specifications will automatically be supported because the algorithm for searching an element's representation is already implemented. Thus, editors can query the representation of elements and visualize all in their implementation supported parts of the representation specification.

7 Application and Evaluation

Context-sensitive multilevel modeling is a technique that can be used in systems engineering. This chapter shows example applications of projects in the space domain. It thereby evaluates benefits and drawbacks of this kind of modeling compared two the classical two level paradigm. With its high demand on constraint mechanisms and restrictive attitude towards flexibility, space engineering suits to demonstrate the technique's applicability for systems engineering. Multilevel modeling's higher degree of flexibility, compared to the two level paradigm, has to be controlled to make it applicable for this domain.

This chapter is structured as follows: the first part shows exemplary applications of context-sensitive multilevel modeling for systems engineering and the second part discusses to what degree it fulfills the requirements specified in Section 1.3.

7.1 Application for Space Projects

The exemplary applications of context-sensitive multilevel modeling aim to show solutions for general challenges of systems engineering. The first example demonstrates the adaptability of a system model to projects of different complexity. The second example shows how multilevel modeling can be used to handle model evolution and reuse of model elements. The last example shows how multilevel models can be used in projects with before unforeseen model levels. Furthermore, the examples show that multilevel modeling can be used in interdisciplinary environments and with domain-specific editors and artifact generation.

Models of these exemplary space projects are implemented in the MBSE tool Virtual Satellite. To compare modeling with and without an application of context-sensitive multilevel modeling, this section presents implementations in Virtual Satellite and in its extended multilevel version, presented in Chapter 6.

Each example is structured as follows: the first part describes how it is modeled in Virtual Satellite and highlights problems without multilevel modeling. The second part describes a solution with context-sensitive multilevel modeling. The last part compares the solutions and discusses benefits and drawbacks.

7.1.1 Varying Complexity in System Models

Modeling environments need to be adoptable to different complex projects. Figure 7.1 shows an example of a satellite with a reaction wheel (RW) as part of the attitude and orbit control systems (AOCS). Reaction wheels can be controlled from ground by using telecommands. To describe the possibility to turn the reaction wheel on remotely, engineers could model a corresponding telecommand.

Problem In the current implementation of the model-based systems engineering tool Virtual Satellite, system models contain trees for different abstraction layers. In a model as shown in Figure 7.1, a tree exists for the layers *ProductDefinition*, *Configuration* and *Assembly*. Thus, even without using

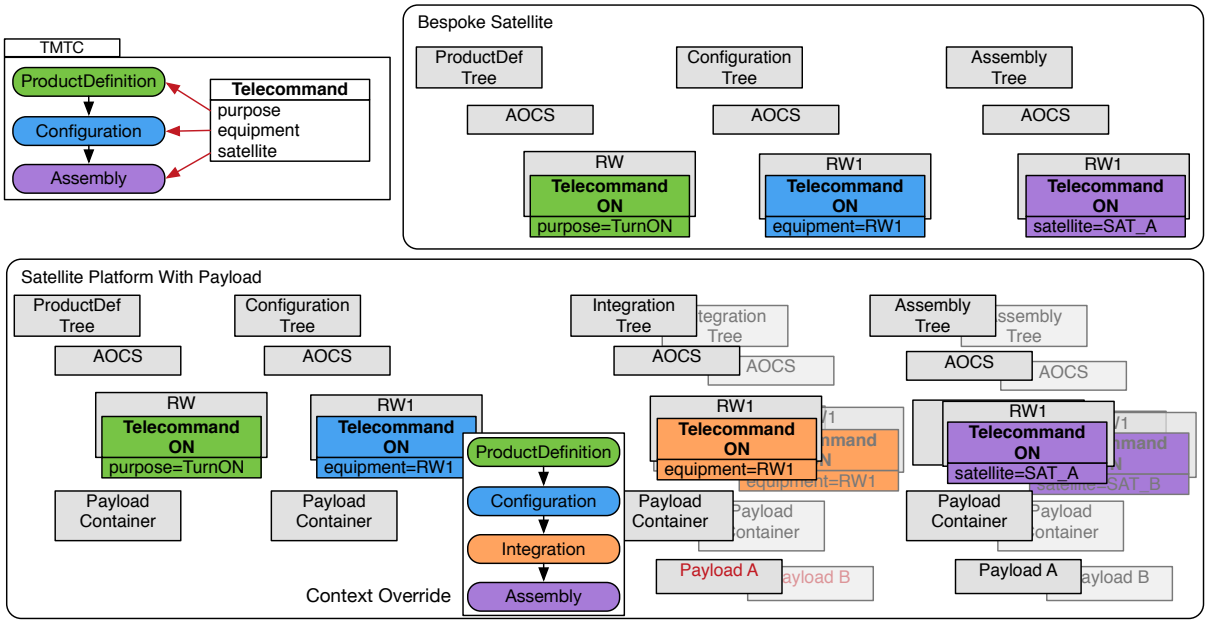

Figure 7.1: Context redefinition can handle different complex projects with varying number of model levels.

multilevel modeling, system models in Virtual Satellite contain different abstraction layers by convention. If a more concrete layer is created, all elements of the higher level are copied. This way, however, model elements are not customized to the different degree of abstraction. If not explicitly modeled for all abstraction layers, a telecommand, e.g., has the properties for the satellite's id even in the first, most generic layer. Thus, because specific information might not be available in some early phases, such model elements might lead to misconceptions, redundant information and conflicts.

Furthermore, different projects might need different layers of abstraction. A system model for a satellite project is not as complex as a project for a satellite platform. Such a platform might require an additional model level for the integration of the payload. If both projects share a base model, it needs to be possible to insert an additional abstraction layer in the model of the platform project.

Context-Sensitive Mulitlevel Solution With context-sensitive multilevel modeling, properties can be mapped to context phases, which describe model levels. To support a model structure as in the regular Virtual Satellite, the context can define model levels that correspond to Virtual Satellite's tree structure. As shown in Figure 7.1, such a context defines phases for *ProductDefinition*, *Configuration* and *Assembly*. The purpose of the telecommand is already known at product definition while the satellite's id can be configured at assembly level only. This way, the telecommand's properties are only visible in phases where its values are known.

However, different complex projects might need varying numbers of process phases. As argued before, the example shows an initial context definition that represents the default abstraction layers of a project modeled in Virtual Satellite. Projects with a payload module might require an additional instantiation step for the payload integration.

One of the benefits of context-sensitive multilevel modeling is that a system's context, which specifies its modeling levels, is maintained separately. If circumstances of the systems engineering pro-

cess change, its context can be adapted. As shown in Figure 7.1, it is possible to redefine the context for projects with, e.g., an additional development step for payload integration. As the integration is done before the final assembly, an additional *Integration* phase has to be added between the existing phases.

Context redefinition can be done either explicit or implicit. Explicitly redefining the context should be done if it is clear that there are several applications of the system model with the new context. This explicit redefinition is done in the context editor and allows to model a new context starting from the current phase. Editing rules are only that all previous phases remain the same and further existing phases still have to be part of the context. Such a redefinition allows to add new phases, change their order or to add new alternative paths. However, to reduce modeling effort, if a project needs an additional instantiation step because of, e.g., changing requirements with resulting changes in only some domains, an implicit redefinition is possible. Such an implicit redefinition instantiates all elements of the previous level and sets the current context phases to a clone of the previous ones. The additional context phase can then be used for logging purposes.

As there might be several satellites developed based on this platform, an explicit redefinition of the context makes sense. All system models of the platform's satellites need to contain the additional integration level. Thus, the redefined context can be maintained with the platform's base system model. Satellites of this platform can then instantiate its system base model and automatically take the redefined context.

Discussion Context-sensitive multilevel modeling allows to customize editing capabilities of elements to the corresponding abstraction level. Furthermore, these customizations and model manipulation constraints can be adopted to changing requirements. This example would not have been supported with potency-based multilevel modeling.

Future work on context-sensitive multilevel modeling could investigate further rules for context redefinition. Changes in the context must not circumvent model manipulation constraints. Furthermore, it might make sense to develop rules for who is in charge of the context and who is allowed to edit it. Such rules, however, are highly application-specific.

7.1.2 Reuse in Systems Engineering

System models develop from abstract ideas to concrete implementations and thereby grow with respect to the information content in the model. Elements modeled in early project phases are instantiated in later phases. Besides this development of the system model, a second dimension of model element reuse exists. External technology that is used in the system model, such as network types, are specified within the model to use them in the system description. While the first dimension represents the engineering progress, the second dimension makes external technology available to use in the system description. An instantiation along the first dimension creates all or most concepts of higher level, whereas instantiating technology is done separately when needed.

Current State and Problem Modeling a technology solution and creating an instance of it in the system model is not supported by the Virtual Satellite editor. To model the application of, e.g., a *DSubBus* in the system, users have to manually create a generic instance, such as a connector element, and then set its type relation to the technology. Also corresponding technology parameters have to be created manually.

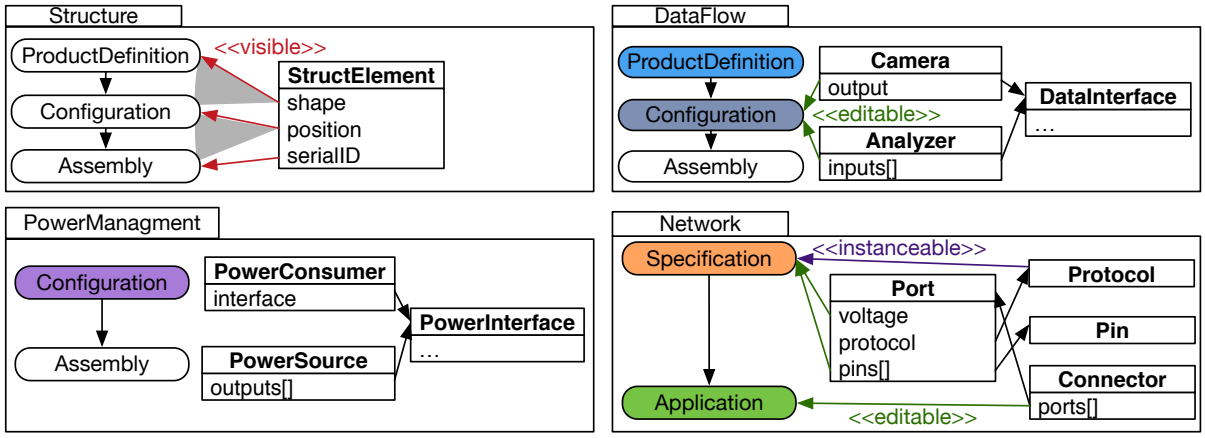

Figure 7.2: Domain concept definition in context-sensitive multilevel modeling. Instances of these elements in Figure 7.3 are filled with the color of the here defined context phases.

Context-Sensitive Multilevel Solution With context-sensitive multilevel modeling, context-based constraints can be used to customize the domain concepts to the different abstraction layers. Figure 7.2 shows the definition of domain elements for a satellite with cameras and image analyzers. Each domain metamodel can have an own context definition. The concept definition of a structural element could be based on the afore mentioned trees, such as *ProductDefinition* and *Configuration*. With constraints as in Figure 7.2, a structural element's position would not be visible in the product definition phase.

Context phases, however, do not necessarily have to be based on these trees, which represent the system model abstraction layers. Instances of the *Network* domain metamodel can be used to specify a technology in any element container and then instantiate these elements in the system description. Context-based constraints for *Ports* specify that its voltage and pins can only be edited in the specification phase. Applications of the network elements in the system model can connect ports with connectors. Besides these editing constraints, the network metamodel constraints instantiation of the protocol element. The *NetworkProtocol* specifies how communication with this network technology works. Because the protocol cannot be changed anymore when using the technology, the protocol can only be instantiated in the specification phase. Furthermore, as shown in the power management metamodel, context models do not necessarily require context-base constraints but can also be used as documentation or to, e.g., create filters for specific phases of a certain domain.

While the context-based constraints are visualized graphically in Figure 7.2, their definition in the multilevel-based Virtual Satellite is done textual. Properties in the extension mechanism are annotated with specifiers, such as `serialID editable at Assembly`. However, this implementation is interchangeable. Specialized tools can also implement graphical editors for context-based constraints.

Figure 7.3 shows a system model that is based on the domain metamodels in Figure 7.2. Figure 7.3 shows the first two abstraction layers of the satellte, a *ProductDefinition* tree and a *Configuration* tree. Besides domain elements, modeled as clabjects, trees are based on containers, which help to structure the model but do not contain domain properties. Containers represent structural elements

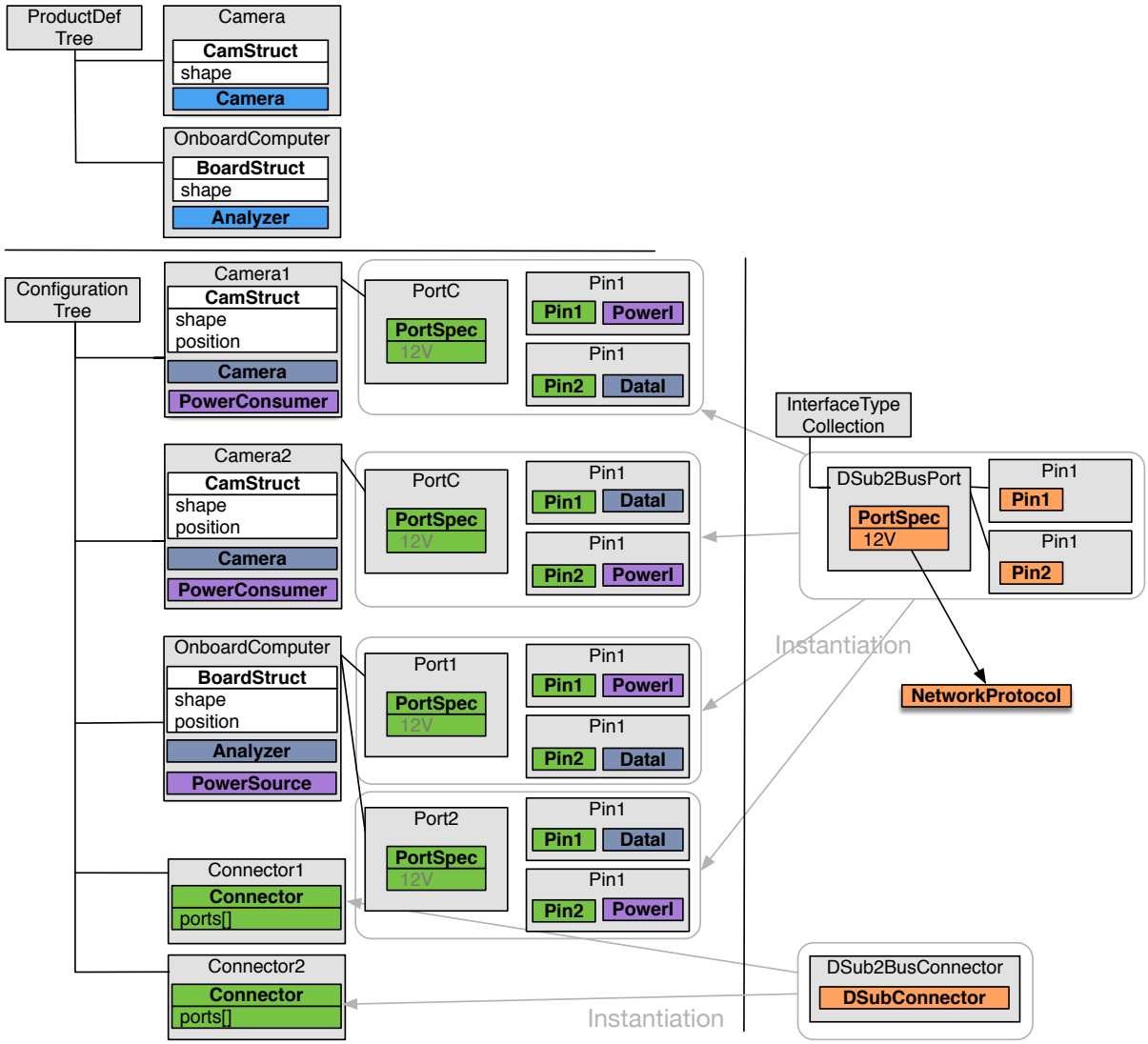

Figure 7.3: Context-sensitive multilevel model for the first two abstraction layers of a satellite design and an addtional modeling level for interface type definition.

and compose the different domain elements. A camera could, e.g., be modeled as container and then hold its property values for wight, power consumption and physical shape in elements of the corresponding domain metamodel.

On first, highest abstraction level the system model describes that a satellite can have cameras and onboard computers. Engineers can create a first basic shape for these elements to visualize them in, e.g., a 3D model. Furthermore, as part of the image analysis domain, camera elements can be defined as image source and the onboard computer as analyzer. This first level does not contain any satellite composition or quantities of elements yet.

The *Configuration* tree instantiates the elements of the *ProductDef* tree and adds more concrete system details. While without multilevel modeling the elements are just copied to the new tree, with context-sensitive multilevel modeling the elements can have different properties depending on the current modeling level. Instantiating elements in the new tree automatically increases the

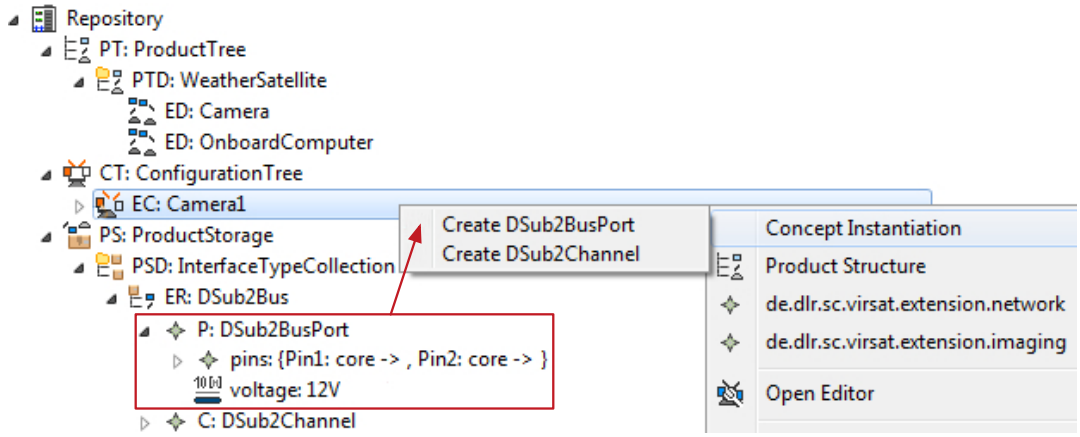

Figure 7.4: Editor support for application of technology. The editor provides an option to instantiate the before modeled technology, as here the *DSub2Bus* elements.

model level to the next phase modeled in the domain element's context. In the *Configuration* tree in Figure 7.3 the system has two cameras connected to an onboard computer. Furthermore, with structural elements in the configuration level, its elements can now configure the position property.

Besides this concretization of the system along abstraction layers, context-sensitive multilevel modeling can also be used to model the instantiation of technology solutions specified along a different modeling dimension. In contrast to classical multilevel modeling, where there is a global order of levels numbered consecutively, in context-sensitive multilevel modeling levels are ordered for different domains separately. Since the context for the network domain metamodel in Figure 7.2 specifies that network elements are in a *Specification* level first and then in an *Application* level, network elements do not fit into the dimension of *ProductDefinition*, *Configuration* and *Assembly*. Instead, network elements can be created in any container element and, as specified in the context, are initially in the *Specification* level. Other container elements, such as trees, can then configure to use this network specification container as type-scope. As shown in Figure 7.3, this enables to instantiate elements from type-scope containers, such as the *InterfaceTypeCollection*. The specification of the *DSub2BusPort*, with its two pins, allows to instantiate it in the *Configuration* tree. As the ports on the cameras and the onboard computer, instantiated elements are then in the *Application* level. The port voltage cannot be changed anymore, but connectors can now be used to connect ports. Because ports and connectors are on *Application* level, the *NetworkProtocol*, which is restricted to the *Specification* level, can not be instantiated inside their containers.

Instantiating the *DSub2BusPort* inside the satellite's configuration level automatically also instantiates the two pins, modeled on specification level. In their application, however, the pin layout can be customized. While on the first connector, connecting *Camera1* and the *OnboardComputer*, the power interface is on the first pin and data on the second, for the second connector it is switched.

As shown in Figure 7.4, the multilevel version of Virtual Satellite editor supports the instantiation of model elements. Using the context menu to create a *DSub2BusPort*, automatically creates the modeled pins.

Discussion Modeling a system with context-sensitive multilevel modeling as shown in Figure 7.3 supports a model structure as currently used for space system engineering. Furthermore, it provides a more consistent implementation of the editing capabilities of the different abstraction layers of the system. Modeling the context and, thus, the modeling levels per domain removes the restriction to have one global linear order of modeling levels. This better maps real world development of systems and technology. Concepts of different systems, domains or technology develop differently and models and engineering processes need to respect this. As a result, context-sensitive multilevel modeling improves interdisciplinary engineering. Furthermore, the generic instantiation operation of multilevel modeling requires less modeling effort because, e.g., network elements have to modeled only once and can then be instantiated in the system.

7.1.3 Unforeseen Instantiations in Follow-up Projects

As a nature of systems engineering, different projects target development of different aspects of a system. With the definition of goals and scope for one project, it is impossible to foresee all further possible activities.

Current State and Problem An example for this challenge is the project autonomous terrain-based optical navigation (ATON), which targets to develop a navigation system for space exploration missions [10]. The software development is based on a model-driven approach with domain-specific editors and code generation [15]. Because of its success, a follow-up project tries to optimize ATON's software for different computer architectures [58]. It, therefore, uses the system model and tries to map ATON's software tasks to processing nodes. This instantiation of software tasks in different computer configurations was not anticipated when the ATON model was designed.

The current model infrastructure of these two projects can be seen in Figure 7.5. As there are several software components used multiple times in the architecture, the ATON model has two explicit levels. The first level is used to model the software components with their inputs and outputs, the second level instantiates these components in the system description and connects them. This instantiation process is based on promotion. Promotion uses one model, here the *Definition Model*, and *promotes* its elements to types for another model [31]. A *CraterNavigation*, modeled in the definition model can then be instantiated, e.g., for different cameras oder planets. As shown in Figure 7.5, this leads to a complex metamodel with elements for each level. Explicit modeling of levels not only bloats the metamodel, it also introduces implementation details within the metamodel. To model the domain aspects as clean as possible and to make it implementation independent, this should be avoided.

Besides these two levels of the initial project, the follow-up project, which maps the software tasks to computing nodes, introduces an additional third level. Because the ATON model design did not consider an additional third level, this task-node mapping is done in a separate modeling environment. As a result, mapping task instances to nodes takes much modeling effort: To describe a configuration, users have to create *TaskConfig*s for each task instance, the configuration should use and set the reference to the corresponding task instance.

Context-Sensitive Mulitlevel Solution Modeling the system shown in Figure 7.5 with context-sensitive multilevel modeling simplifies the metamodel and removes implementation related elements. The *Tasking* metamodel, shown in Figure 7.6, contains a level-independent definition of the tasking elements. Context-based constraints allow to specify inputs and outputs of tasks in the first, *TaskDefini-*

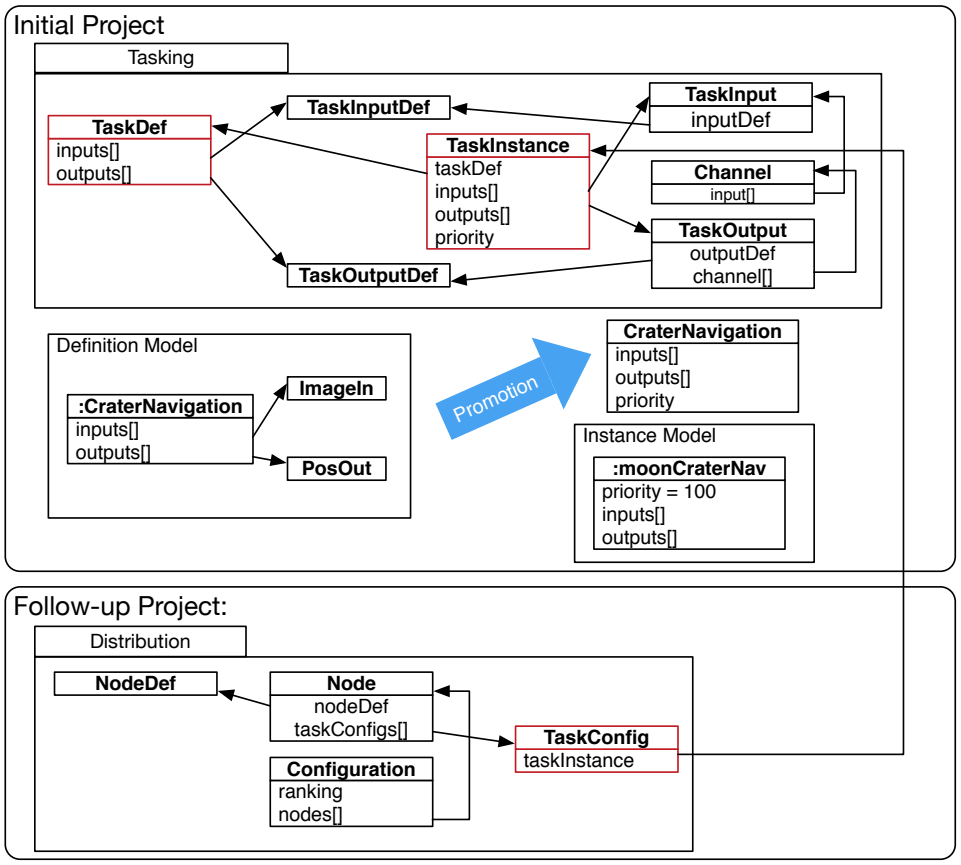

Figure 7.5: Unforeseen Instantiation in Follow-up Projects. Promotion in the initial project allows to instantiate dynamically modeled software tasks. A follow-up project creates a third, previously not anticipated, model level.

tion, level only. Connecting tasks and channels can only be done on *Application* level. This context-sensitive metamodel thereby corresponds to the tasking metamodel in Figure 7.5. In addition to that, it allows to specify custom channel types, with dynamic parameters, in the first modeling level. Extending clabjects with new field definitions is only possible if explicitly allowed, as for *Channel*s in this example.

On *TaskDefintion* level, the system model in Figure 7.6 specifies tasks for sensors with only outputs, such as a camera, and analyzer components with inputs. Besides tasks, this level also creates a first-in-first-out implementation of a channel. Utilizing the extensibility of channels at this level, it specifies an additional parameter for its buffer size. The implementation of the domain-specific editor for this first level is based on a textual language for element definition. As the tool environment's extension mechanism, this *TaskDefinition*-language can use the context-base-language to specify context-based constraints for new fields. An editing constraint for the *bufferSize* could be `bufferSize editable at Application`.

The project's next level then instantiates the tasks and the *FIFOChannel* and connects these, previously defined, elements. The *Application* level's editor uses a graphical domain-specific language to visualize the connections of the elements. In this editor, the instantiation-mechanism of the

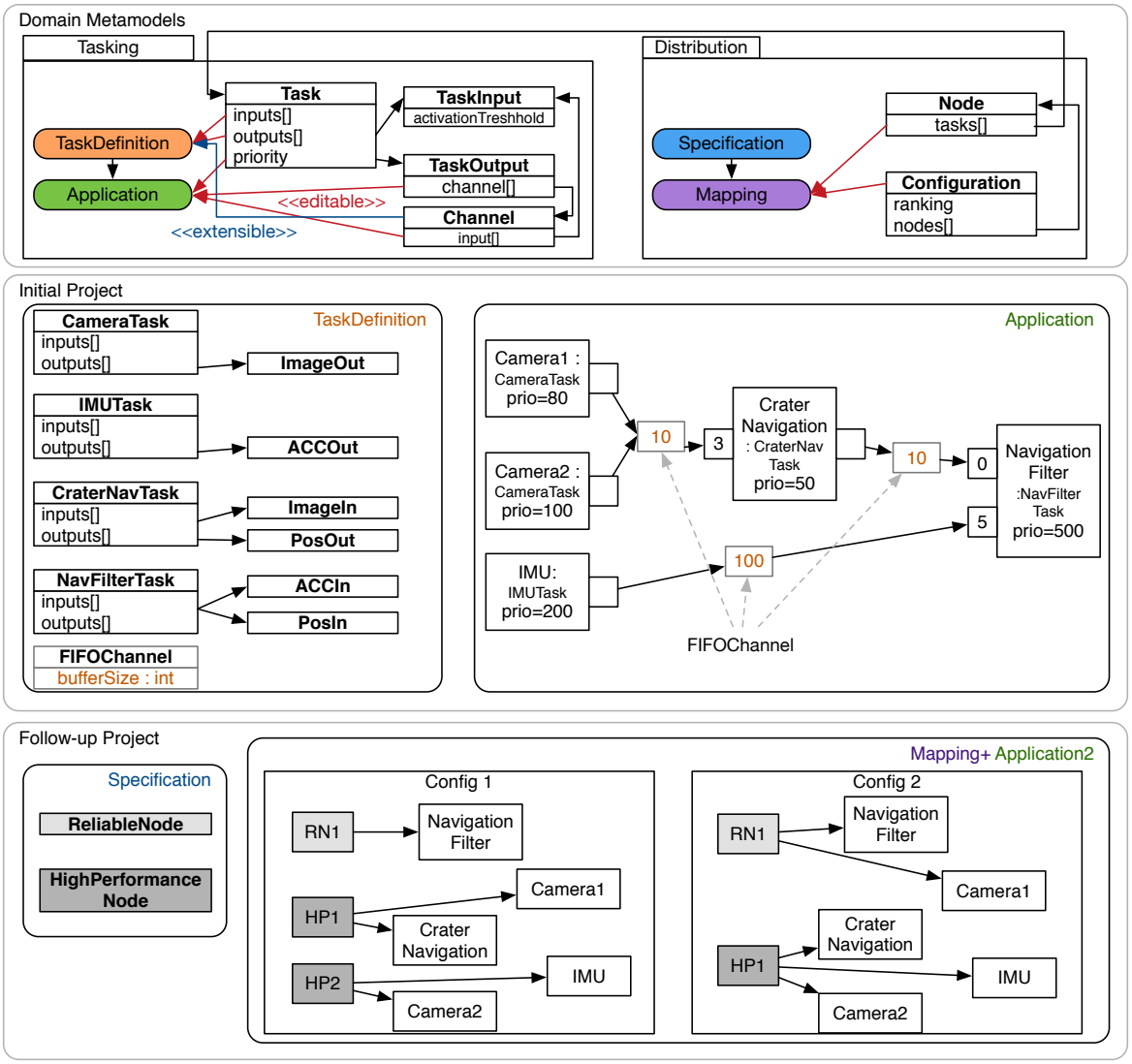

Figure 7.6: Context-sensitive multilevel solution for unforeseen instantiation levels in systems engineering. Implicit context redefinition allows to use the tasking elements in a third level.

multilevel environment can be used to instantiate the tasks, defined on *TaskDefinition* level, and automatically add their inputs and outputs. As specified in the *Tasking* metamodel, it is not possible to add new inputs our outputs to tasks at this level. In the graphical language, inputs and outputs are rendered as ports. Users can connect them with channels to specify the system's data flow. Furthermore, it is possible to assign priorities to tasks and an activation threshold to inputs. As defined on the *TaskDefinition* level, *FIFOChannels* have to specify a buffer size.

The application level of the system model in Figure 7.6 shows two cameras sending images into a shared *FIFOChannel*, which transfers the data to a crater navigation module. This crater navigation estimates the current position and sends it to the *NavigationFilter*. Furthermore, this filter analyzes input data of an inertial measurement unit (IMU). Because the IMU produces data with a higher frequency than the cameras, the channel's buffer size is larger for the IMU data than for the image data.

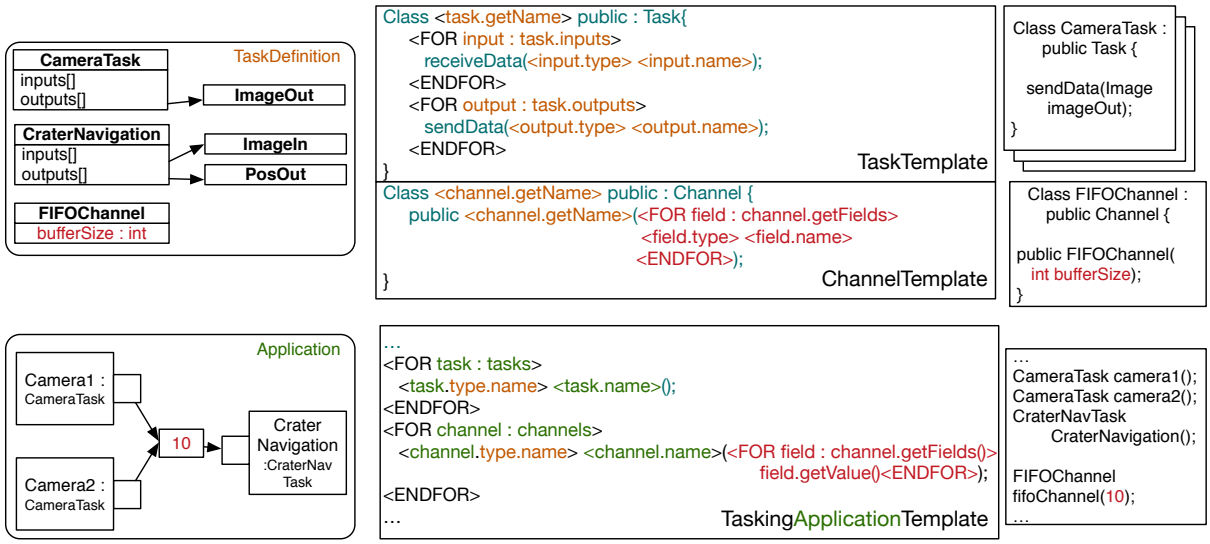

Figure 7.7: Artifact generation from multilevel models with domain-specific infrastructure. The left part shows the model, the middle part code templates and the right part shows the resulting source code.

The second domain metamodel for the follow-up project, which maps tasks to processing nodes, has also two model levels. The first level models existing computing nodes, which can then be used in the configurations on the next level. To instantiate tasks in this new, third model level, the tasking context has to be redefined. This context redefinition corresponds to an adjustment of the system model to the new project scope of the follow-up project. An implicit redefinition of the context allows to add a second, cloned *Application* level, which is then called *Application2*, without further modeling effort. The resulting model level connects the *Mapping* phase of the *Distribution* domain with the new, implicit *Application2* phase of the *Tasking* domain.

The system model's first configuration uses one reliable node for the *NavigationFilter* and two *HighPerformanceNode*s for the rest of the tasks. The second configuration specifies how to handle a failure of one of the high performance nodes.

Figure 7.7 shows how code templates can be used to generate source code from models. The first model level is used to generate classes for tasks and channels, the second level describes how these classes are instantiated as objects. The definition of tasks with their inputs and outputs is used to generate methods for sending and receiving data.

Besides tasks, the first tasking level is used to generate channel classes. With the definition of channels being extensible in the *TaskDefinition* level, it makes sense to also consider new field definitions of channel specifications. Thus, the channel template creates a constructor with a parameter for each field property. Object instances can then configure the channels by passing the model's property values as constructor parameter. The channel size of *10*, specified on the *Application* level for the only shown channel in Figure 7.7, results as an integer value of the *fifoChannel*'s constructor. This way it is possible to even use dynamically added clabject fields in source code generation. Besides object initialization, it is possible to also generate code for communication. The tasks' methods to send and receive data can be used to generate code for data transportation as described

by the connections in the model.

Discussion Modeling projects, which require multiple model levels to describe relevant aspects of a system, benefit from context-sensitive multilevel modeling. Model complexity is reduced compared to solutions without multilevel modeling. If model levels are not modeled explicitly, the model is less technology dependent. Furthermore, the multilevel modeling environment's instantiation mechanism can be used instead of implementing a separate instantiation mechanism for each level, as with the promotion for task definitions. Additionally, with a unified handling of ontological types, elements are natively prepared for additional, previously not anticipated instantiation levels of, e.g., follow-up projects. A project as the task-node mapping for ATON, which instantiates previously defined software components, does not require the definition of further, infrastructure-related, model elements or instantiation mechanisms.

The type extension management, which allows to add new fields to clabjects only if explicitly specified, enables to use the flexibility of multilevel modeling, if necessary, but in a controlled way. Furthermore, it supports developing model-based tools and transformations: the design decision that channels are extensible at the *TaskDefinition* level, and only then, allows to explicitly consider this flexibility in editors and code generators. The channel templates for code generation translate channel field definitions to implementation parameters.

Separate editors for model levels can be completely different. The *TaskDefinition* level, which mainly defines software tasks, uses a textual syntax in the Virtual Satellite implementation but could also be based on a class-diagram related graphical representation as shown in Figure 7.6. The *Application* level, on the other hand, use a graphical language to visualize the communication and data flow of the software. The implementation of these highly customized editors is only possible because of the domain-specific infrastructure, which is generated from the domain metamodels.

7.2 Evaluation of Requirements

The goal of this thesis is to find new modeling concepts for systems engineering that support dynamic type instantiation. To ensure applicability for space systems engineering, Section 1.3 introduced a set of requirements. This section evaluates if context-sensitive multilevel modeling fulfills these requirements.

Requirement REQ.1: Dynamic Type Instantiation The first requirement demands that modeled concepts can be instantiated in other parts of the system description. As shown in all previous examples, this is possible. A modeled network port can be used on several different locations in the system description. A modeled camera driver can be used for all cameras used in the system. The corresponding ontological type-instance relation is handled uniformly by using clabjects for domain concepts.

Requirement REQ.2: Model Manipulation Constraints Constraints for model manipulations and element instantiations can be specified using a new multilevel context. Context-based constraints can express the same restrictions as concepts of classical multilevel modeling, such as potency or durability, but are more flexible and can be understood by domain experts without modeling experience. The definition, in Figure 7.2, that a network port's voltage is only editable in the specification phase prevents changes in the application phase. Thus, with context-based model manipulation constraints the environment fulfills Requirement REQ.2.

Requirement REQ.3: Adaptability to Different Complex Projects Requirement REQ.3 demands that the model language and the model itself is adoptable to different complex projects. As shown in Section 7.1.1, this is possible with context-sensitive multilevel modeling. The implementation of domain elements as clabjects unifies the handling of ontological typing and, thus, allows an arbitrary number of model levels. Context redefinitions can be used to adjust the system model and its context to varying number of instantiation levels. Model manipulation constraints keep targeting their intended level even if new model levels are added between two existing ones.

Requirement REQ.4: Interdisciplinary Engineering Multilevel models can have elements of an arbitrary number of different domains. The example in Section 7.1.2 shows how elements of different domain metamodels can be combined. Furthermore, as model levels are considered by order-alignment and for elements of different domains separately, their combination does not cause problems. The definition of elements in a domain metamodel, additionally, allows to create domain-specific editors for the different domains. Models can then be edited in parallel by editors of different domains, showing only the relevant elements for the current user. With the mapping of multilevel elements to process phases, it is also possible to customize elements according to the current abstraction level. Editors can then filter elements that are not relevant in the current development process phase. Thus, context-sensitive multilevel modeling fulfills Requirement REQ.4 by not only allowing interdisciplinary engineering but also supporting it.

Requirement REQ.5: Domain-Specific Representation Requirement REQ.5 demands that model elements and their editor representation can be customized. Multilevel modeling allows to customize the representation of an element's instances by describing them on higher ontological model levels. For such a dynamic element representation specification, the here presented multilevel modeling environment provides a model language that can be used to attach specifications to generic model elements. Besides this dynamic element customization, the domain metamodel's infrastructure generation can be used to develop completely new model editors. The example in Section 7.1.3 shows how such domain-specific editors can look like. The definition of tasks and channels can be done either, as shown in Figure 7.6, by using a representation similar to class diagrams or by using a textual extension language. To visualize the software's data flow, the editor for connecting tasks and channels uses a graphical domain-specific language. Thus, models using context-sensitive multilevel modeling can have domain-specific representations and Requirement REQ.5 is fulfilled.

Requirement REQ.6: Artifact Generation Multilevel models can be source to generate project artifacts. As shown in the example in Section 7.1.3, templates can be used to, e.g., generate source code from the multilevel models. With domain-specific infrastructure, generated from domain metamodels, templates can even access model elements by using generated getters. Dynamic fields, which are not specified in any domain metamodel, can be accessed by using the generic clabject infrastructure. As shown in Figure 7.7, different model levels can either be used with different templates, dedicated to specific levels, or in one template by following an element's reference to its ontological type. Section 5.4.2, furthermore, shows how code generation can be triggered from different model levels. As a result, Requirement REQ.6 is fulfilled, artifact generation from models with several levels is possible.

Requirement REQ.7: Dynamic Type Extensions Requirement REQ.7 states that type extensions need to be restricted. Thus, with the possibility to instantiate clabjects in multilevel environments, their

extension needs to be constrained. The, in this thesis presented, implementation of a multilevel modeling environment for systems engineering restricts the definition of new fields for clabjects by default. New fields can only be added if explicitly allowed for corresponding elements in the current context phase. The properties of the domain elements in the examples of Section 7.1 were defined in their domain metamodel. The only dynamically defined field is the *bufferSize* of *FIFOChannel*s in the third example. This dynamic extension is possible because the domain metamodel specifies that channels are extensible in the first model level. Thus, by default, clabjects cannot be extended with new property definitions. To fulfill Requirement REQ.7, such dynamic extensions have to be explicitly allowed by context-based constrains.

8 Discussion

Context-sensitive multilevel modeling is applicable for systems engineering. While the last chapter demonstrated that it can be used even in the strict space systems engineering, this chapter discusses its drawbacks and benefits compared to other multilevel modeling approaches in literature. Furthermore, it highlights why it is necessary to consider the model's context when using multilevel modeling in systems engineering. The last part of this chapter discusses aspects of context-sensitive multilevel modeling that open new research potential.

8.1 Multilevel Modeling Tool Environments

One of the most sophisticated tools for multilevel modeling is the tool Melanee [49]. The multilevel environment, presented in this thesis, is based on several of Melanee's multilevel concepts, such as the clabject. Melanee, furthermore, provides means to dynamically customize element presentation by even combing textual and graphical domain-specific languages [36]. While this work's environment implementation also provides basic means for dynamic element representation specification, its focus lies on domain-specific infrastructure to support new domain-specific tools, developed from scratch. A tool specifically developed for a domain or project can fits its requirements better than a generic tool which is customized to the domain.

As context-based constraints in context-sensitive multilevel modeling, Melanee features a dialect of the object-constraint language (OCL) to specify level-spanning constraints, which are aware of the ontological modeling dimension [59]. Underlying concepts, such as potency and durability, however, are based on a stiff level naming scheme that uses consecutively numbered labels and, thus, does not support to insert new levels between others. Multilevel-objects, called *m-objects*, solve this problem by labelling levels with unique names [34]. Concretization hierarchies describe the order of these levels and allow to insert new ones without changing the others. Context-sensitive multilevel modeling is based on this approach and goes one step further by modeling a context, which describes the order of levels. Contexts can have domain-specific representations, as e.g., a process model for systems engineering, and allow to adjust models according to their application contexts.

Context-sensitive multilevel modeling is in line with Frank's ideas of multilevel models for language hierarchies [30]. Top-level languages can be used for whole domains, more concrete languages for organizations and finally most concrete languages for projects or specific systems. As Figure 8.1 shows, it can supports it by customizing the model and its languages to the requirements of organizations or projects. In the figure, the space agency defines some high level elements that can be used in all space systems and specifies a basic, initial context. The satellite devision adds a level for specialized satellite elements. The development of a satellite platform configures the system model and adds an additional context level for payload integration. This way, elements of the first level can be instantiated in different complex projects and it is possible to, e.g., specify that a parameter is only editable in the assembly phase regardless of how many levels are before it.

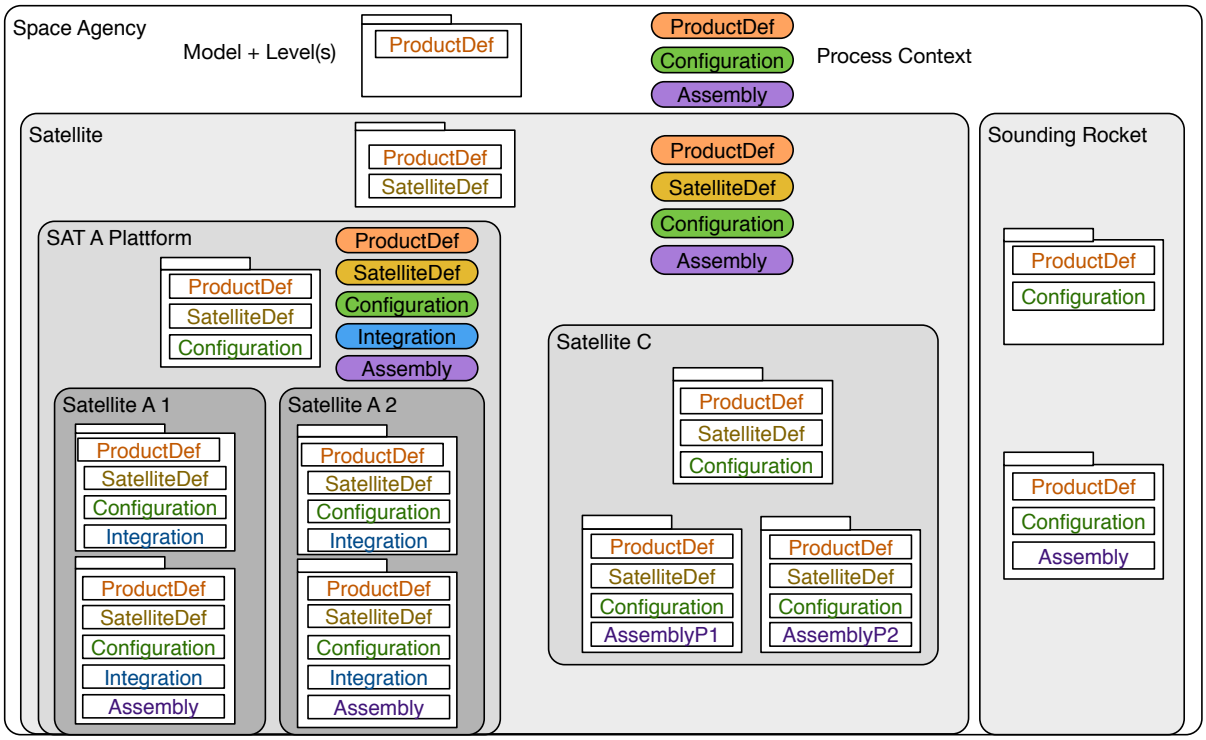

Figure 8.1: Context-sensitive multilevel model hierarchy in organizations. Organizational units or projects can adjust the context and add new model levels to the model.

8.2 Multilevel Modeling for Systems Engineering

Multilevel modeling is a technology with enormous potential. Its increased flexibility, compared to the two level modeling paradigm, and its element evolution from abstract to concrete are beneficial for systems engineering. Model-based systems engineering, however, has strict requirements on models. Flexibility has to be controllable to prevent conflicts and misconceptions. Models have to be adjustable to changing requirements and modeling concepts have to be understood by domain experts.

Assigning potencies to model elements requires to know the next instantiation levels. As shown in Figure 8.1, this is not always possible for systems engineering. Depending if, e.g., a generic camera of the *ProductDef* level is instantiated in a sounding rocket or a satellite, a different number of instantiations is necessary. In a sounding rocket's system model, a camera instance could be a configured system element, whereas camera instances in a satellite division model could be just cameras, with specialized characteristics, in a satellite element inventory.

In contrast to classical model manipulation or instantiation constraints, context-based constraints allow to specify that elements are editable in specific levels only. Field durability only restricts until which level a field can be edited. Specifying that a property should be editable only after a specific level is not possible. For systems engineering, however, such constrains are required. If a serial id, for example, can be set before the assembly level then an earlier configured placeholder value could make an engineer falsely think that the value is already set.

System models combine elements of different domains. Furthermore, as Section 7.1.2 highlights,

in system models there exist different reuse and instantiation dimensions. Thus, a consecutively numbered, global level labeling scheme is to rigid. System models do not have a global, linear order of model levels. Instead, level order and naming should be considered for different domains separately. Recent publications in this domain, such as about the local total order alignment, show the relevance of this topic [55]. Since a global order of levels is not feasible for systems engineering, context-sensitive multilevel modeling considers levels for elements of different domains separately.

In model-based systems engineering, models are the central storage of knowledge. As a result, models are not only accessed and edited by modeling experts but also by experts of different domains. Asking a domain expert how often a new domain element can be instantiated is not likely to be successful. On the other hand, asking when, in the modeled process, an element can be edited can probably be answered. Context-based constraints, thus, allow non-experts in modeling to understand these model mechanisms.

As multilevel modeling tools are usually implemented on top of two level environments, there is no domain-specific model infrastructure available. This is a critical loss, compared to the two level paradigm, because model-based system engineering is most beneficial with project-specific tool-support. Implementing tools without domain-specific infrastructure, such as element getter-methods, takes more effort and is error prone. To use multilevel modeling and still have a domain-specific infrastructure, it is possible to specify domain elements in a dedicated domain metamodel of a third typing dimension. This third typing dimension can then be used to generate infrastructure.

Concluding, compared to other domains, multilevel modeling for system engineering faces additional challenges:

- Exact number of levels required to describe a system is not predictable
- The global level-numbering scheme is to rigit
- Model manipulation constraints have to be understood by domain experts
- Domain-specific infrastructure is necessary for project-specific tools

Context-sensitive multilevel modeling, as described in this thesis, shows solutions to these challenges and, thus, makes multilevel modeling applicable for systems engineering. There is, however, potential for further research on this kind of modeling.

8.3 Future Work on Context-Sensitive Mulitlevel Modeling

Context-sensitive multilevel modeling's separation of domain elements to contexts, which explain their level order, corresponds to Kühne's separation of elements and their classification hierarchies to *modeling spaces* [55]. Future work can investigate potential conflicts of relationships between elements of different domains, respectively modeling spaces. Especially relationships which require a targeted element to be in a specific level might be interesting.

This work introduced process models as context description. Future work could investigate other forms of context to describe the order of levels. The use of models for language hierarchies in organizations, as described by Frank [30], e.g., suggest to describe organizational structures in a context model.

Another interesting prospect of context models is how a global combination of contexts with elements from different domains can be used. As context models describe how system elements evolve in the development process, the current state of system elements in their context can give an overview of the current development activities.

Besides different forms of context description, future work could search for rules of context redefinition. Such rules should answer the trade-off of model adoptability and editing constraints. A complete redefinition of the context could lead to infiltration of the context-based constraints, without redefinition, on the other hand, models are not adoptable to changing requirements.

To improve the representation of dynamically added elements, the modeling language for the element presentation specification of instances could be extended. A domain-specific language could specify editor customizations, such as how an element is visualized in a graphical language or what kind of textual representation it has. Furthermore, it could also describe user interface options, as what kind of input fields are visible. As, e.g., the tool Melanee has advanced customization methods, it might make sense to combine both approaches.

With the definition of elements of one domain in a dedicated domain metamodel, future work can investigate how updates of such a metamodel can be handled. As all instances of domain elements are serialized as clabject, which is defined in the static linguistic metamodel, an update of the domain element does not necessarily require a system model to be transformed. Thus, updating the domain metamodel and the context can be used to generate infrastructure for properties of a later added, additional instantiation layer.

Besides research on context-sensitive multilevel modeling itself, future work could examine the integration of context-sensitive modeling into existing multilevel modeling tools.

9 Conclusion

Technology and systems develop from abstract ideas to concrete implementations. In this process, existing concepts are step by step used to develop more complex, specialized concepts.

This thesis describes how multilevel modeling can be used to map this evolution and support engineering processes. Multilevel modeling introduces generic elements that can be instantiated in more concrete model levels to describe new elements. With a unified instantiation operation, models can have an arbitrary number of instantiation levels.

Model-based systems engineering uses models as central storage of knowledge. To prevent misconceptions and conflicts, systems engineering requires strict rules for model manipulations. Depending on the application context of the model, different parts can be edited. One model might be used as basis for several projects with potentially varying complexity. Changing requirements in the development might introduce the need to update the model and its editing rules.

As changes on multilevel models can have consequences on several abstraction levels, using this kind of modeling for systems engineering requires level-aware editing constraints. This thesis describes how a separate context model can be used to specify how a multilevel model can be edited. Context models, therefor, describe how model levels are arranged. For systems engineering, the context can be, e.g., a process model. Context-based constraints can then restrict edits to specific process phases. If the application context of a system model changes, the context model can be updated.

Such a context model in combination with domain metamodels, which describe system elements, can be used generate domain-specific infrastructure for external tools, such as editors and artifact generators. Elements from different domains can have different context models. Thus, a system model, which combines several domains, can not have a global linear order of model levels. Instead the mapping of elements to levels is done separately for different domains.

This thesis evaluates context-sensitive multilevel modeling by applying it to systems engineering projects of the space domain. It shows an application of varying complexity and changing requirements. Furthermore, it demonstrates how elements of different domains can be combined in one system model.

Context-sensitive multilevel modeling presents solutions for challenges of modeling in systems engineering. While constraints enable to restrict instantiation and editing capabilities of elements, it is still possible to introduce new abstraction levels. Context models allow to describe how elements are expected to evolve in the model's current application context. As such a description can be based on the element's domain, manipulation constraints are also understandable for domain experts. A separation of application and context model enables to adjust the application model to different projects and changing requirements by updating the context model. An updated context model adjusts constraints to the new requirements and preserves its initial purpose. Thus, this work makes multilevel modeling's potential available to systems engineering while keeping its increased flexibility manageable.

Bibliography

[1] A. L. Ramos, J. V. Ferreira, and J. Barceló. "Model-based systems engineering: An emerging approach for modern systems". In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 42.1 (2012), pp. 101–111. ISSN: 10946977. DOI: 10.1109/TSMCC.2011.2106495.

[2] J. A. Estefan. "Survey of Model-Based Systems Engineering (MBSE) Methodologies". In: *Incose MBSE Focus Group 25.8* 25.8 (2008). ISSN: 0163-6804. DOI: 10.1109/35.295942.

[3] D. C. Schmidt. "Model-Driven Engineering". In: *IEEE Computer* 39.2 (2006), pp. 25–31. ISSN: 00189162. DOI: 10.1109/MC.2006.58.

[4] G. J. Holzmann. "Landing a Spacecraft on Mars". In: *IEEE Software* 30.2 (2013), pp. 83–86. ISSN: 07407459. DOI: 10.1109/MS.2013.32.

[5] C. Atkinson and T. Kühne. "Model-Driven Development: A Metamodeling Foundation". In: *IEEE Computer Society* (2003).

[6] B. Meyer. "Generic Model-Based Systems Engineering Methodology". In: *INCOSE International Symposium* 24.s1 (2014), pp. 207–222. ISSN: 2334-5837. DOI: 10.1002/j.2334-5837.2014.00017.x. URL: http://dx.doi.org/10.1002/j.2334-5837.2014.00017.x.

[7] ESA–ESTEC. *ECSS–E–10A: Space engineering*. 1996. URL: https://elibrary.gsfc.nasa.gov/_assets/doclibBidder/tech_docs/ECSS-E-10A.pdf.

[8] P. M. Fischer, D. Lüdtke, C. Lange, F. C. Roshani, F. Dannemann, and A. Gerndt. "Implementing model-based system engineering for the whole lifecycle of a spacecraft". In: *CEAS Space Journal* 9.3 (2017), pp. 351–365. ISSN: 18682510. DOI: 10.1007/s12567-017-0166-4.

[9] J. C. Mankins. "Technology Readiness Levels". 1995.

[10] S. Theil, N. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, and J. Wohlfeil. "ATON (Autonomous Terrain-based Optical Navigation) for exploration missions: recent flight test results". In: *CEAS Space Journal* 10.3 (2018), pp. 325–341. ISSN: 18682510. DOI: 10.1007/s12567-018-0201-0. URL: https://doi.org/10.1007/s12567-018-0201-0.

[11] F. Andert, N. Ammann, and B. Maass. "Lidar-Aided Camera Feature Tracking and Visual SLAM for Spacecraft Low-Orbit Navigation and Planetary Landing". In: *Advances in Aerospace Guidance, Navigation and Control*. Springer International Publishing, 2015, pp. 605–623. ISBN: 978-3-319-17518-8. URL: http://elib.dlr.de/96323/.

[12] S. Winkler and J. von Pilgrim. "A survey of traceability in requirements engineering and model-driven development". In: *Software and Systems Modeling* 9.4 (2010), pp. 529–565. ISSN: 16191366. DOI: 10.1007/s10270-009-0145-0.

[13] C. Atkinson and T. Kühne. "The Essence of Multilevel Metamodeling". In: *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Vol. 2185. JANUARY. 2001, pp. 134–148. ISBN: 978-3-540-42667-7. DOI: 10.1007/3-540-45441-1. URL: http://www.springerlink.com/content/2lqcmbcn9l2v465r.

[14] J. Pardillo. *A systematic review on the definition of UML profiles*. Vol. 6394 LNCS. PART 1. 2010, pp. 407–422. ISBN: 3642161448. DOI: 10.1007/978-3-642-16145-2_28.

[15] T. Franz, D. Lüdtke, O. Maibaum, and A. Gerndt. "Model-based software engineering for an optical navigation system for spacecraft". In: *CEAS Space Journal* 0123456789 (2017). ISSN: 1868-2502. DOI: 10.1007/s12567-017-0173-5. URL: http://link.springer.com/10.1007/s12567-017-0173-5.

[16] R. Oshana. "Software Engineering of Embedded and Real-Time Systems". In: *Software Engineering for Embedded Systems*. First Edit. Elsevier Inc., 2013, pp. 1–32. ISBN: 9780124159174. DOI: 10.1016/B978-0-12-415917-4.00001-3. URL: http://dx.doi.org/10.1016/B978-0-12-415917-4.00001-3.

[17] B. Cole, G. Dubos, P. Banazadeh, J. Reh, K. Case, Y. F. Wang, S. Jones, and F. Picha. "Domain-Specific Languages and Diagram Customization for a Concurrent Engineering Environment". In: (2013). ISSN: 1095323X. DOI: 10.1109/AERO.2013.6497134.

[18] B. Weps, D. Lüdtke, T. Franz, O. Maibaum, T. Wendrich, H. Müntinga, and A. Gerndt. "A Model-driven Software Architecture for Ultra-cold Gas Experiments in Space". In: *Proceedings of the 69th International Astronautical Congress*. 2018, pp. 1–10.

[19] B. Kennel. "A Unified Framework for Multi-Level Modeling". PhD thesis. University Mannheim, 2012.

[20] A. Kossiakoff, W. N. Sweet, S. J. Seymour, and S. M. Biemer. "Systems Engineering and the World of Modern Systems". In: *Systems Engineering Principles and Practice, Second Edition*. John Wiley & Sons, Inc., 2011, pp. 3–26. ISBN: 9780470405482. DOI: 10.1002/9781118001028.

[21] B. H. Rao, K Padmaja, and P Gurulingam. "A Brief View of Model based Systems Engineering Methodologies". In: *International Journal of Engineering Trends and Technolog* 4.8 (2013), pp. 3266–3271.

[22] A. W. Brown. "Model-Driven Architecture: Principles and Practice". In: *Software and Systems Modeling*. Springer Verlag, 2004, pp. 314–327.

[23] B. Selic. "The Pragmatics of Model-Driven Development". In: *IEEE Software* 20.5 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[24] German Aerospace Center. *Virtual Satellite Website*. (Visited on 09/18/2018).

[25] Orbital Sciences Corporation. *Satellite Image*. URL: https://pics-about-space.com/low-earth-orbit-satellite-communication?p=1#img3644749585287082667 (visited on 09/18/2018).

[26] I. Galvão and A. Goknil. "Survey of traceability approaches in model-driven engineering". In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*. 2007, pp. 313–324. ISBN: 0769528910. DOI: 10.1109/EDOC.2007.4384003.

[27] L. Baker, P. Clemente, B. Cohen, L. Permenter, B. Purves, and P. Salmon. "Model Driven System Design Working Group: FOUNDATIONAL CONCEPTS FOR MODEL DRIVEN SYSTEM DESIGN". In: *INCOSE International Symposium* 6.1 (1996), pp. 1179–1185. ISSN: 23345837. DOI: 10.1002/j.2334-5837.1996.tb02139.x. URL: http://doi.wiley.com/10.1002/j.2334-5837.1996.tb02139.x.

[28] R. F. Paige, N. Matragkas, and L. M. Rose. "Evolving models in Model-Driven Engineering: State-of-the-art and future challenges". In: *Journal of Systems and Software* 111 (2016), pp. 272–280. ISSN: 01641212. DOI: 10.1016/j.jss.2015.08.047.

[29] C. Atkinson and T. Kühne. "Reducing accidental complexity in domain models". In: *Software and Systems Modeling* 7.3 (2008), pp. 345–359. ISSN: 16191366. DOI: 10.1007/s10270-007-0061-0.

[30] U. Frank. "Multilevel Modeling". In: *Business & Information Systems Engineering* 6.6 (2014), pp. 319–337. ISSN: 2363-7005. DOI: 10.1007/s12599-014-0350-4. URL: http://link.springer.com/10.1007/s12599-014-0350-4.

[31] J. de Lara, E. Guerra, and J. S. Cuadrado. "When and How to Use Multilevel Modelling". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (2014), 12:1–12:46. ISSN: 1049-331X. DOI: 10.1145/2685615. URL: http://doi.acm.org/10.1145/2685615.

[32] C. Atkinson and T. Kühne. "Rearchitecting the UML infrastructure". In: *ACM Transactions on Modeling and Computer Simulation* 12.4 (2002), pp. 290–321. ISSN: 10493301. DOI: 10.1145/643120.643123.

[33] B. Neumayr, M. Schrefl, and B. Thalheim. "Modeling Techniques for Multi-Level Abstraction". In: *The Evolution of Conceptual Modeling*. Ed. by R. Kaschek and L. Delcambre. Vol. 6520. Springer Berlin Heidelberg, 2011, pp. 68–92. DOI: https://doi.org/10.1007/978-3-642-17505-3_4.

[34] B. Neumayr, K. Grün, and M. Schrefl. "Multi-Level Domain Modeling with M-Objects and M-Relationships". In: *Conference on Conceptual Modelling* 96.Apccm (2009). ISSN: 14451336.

[35] P. M. Fischer. "Potential of Multi Level Modelling in Model Based Systems Engineering". In: *Dagstuhl Seminar 17492*. 2017.

[36] C. Atkinson and R. Gerbig. "Harmonizing Textual and Graphical Visualizations of Domain Specific Models". In: *Proceedings of the Second Workshop on Graphical Modeling Language Development - GMLD '13*. Montpellier: ACM, 2013. ISBN: 9781450320443. DOI: 10.1145/2489820.2489821.

[37] B. Volz, M. Zeising, and S. Jablonski. "The Open Meta Modeling Environment". In: *... 2011 Workshop on Flexible Modeling ...* (2011), pp. 1–4.

[38] V. Viyović, M. Maksimović, and B. Perišić. "Sirius: A rapid development of DSM graphical editor". In: *INES 2014 - IEEE 18th International Conference on Intelligent Engineering Systems, Proceedings*. 2014, pp. 233–238. ISBN: 9781479946150. DOI: 10.1109/INES.2014.6909375.

[39] C. Atkinson, R. Gerbig, and C. Tunjic. "A Multi-Level Modeling Environment for SUM-Based Software Engineering". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO 2013*. Montpellier, France, 2013. ISBN: 9781450320702. DOI: 10.1145/2489861.2489868.

[40] C. Atkinson and T. Kühne. "Reducing accidental complexity in domain models". In: *Software and Systems Modeling* 7.3 (2007), pp. 345–359. ISSN: 16191366. DOI: 10.1007/s10270-007-0061-0.

[41] H. Lieberman. "Using protypal objects to implement shared behaviour in object oriented systems". In: *Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Portland, OR, 1986.

[42] E. Gamma, R. Helm, J. Ralph, and J. Vlissides. "Creational Patterns". In: *Design Patterns - Elements of Reusable Object-Oriented Software*. Edition 1. Addison-Wesley Professional, 1994. Chap. 3, pp. 94–155.

[43] G. Richards, S. Lebresne, B. Burg, and J. Vitek. "An analysis of the dynamic behavior of JavaScript programs". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10* (2010), p. 1. ISSN: 03621340. DOI: 10.1145/1806596.1806598. URL: http://portal.acm.org/citation.cfm?doid=1806596.1806598.

[44] A. Borning. "Classes versus Prototypes in Object-Oriented Languages". In: *Proceedings of 1986 ACM Fall joint computer ...* 96 (1986), pp. 1–8. ISSN: 1524-4725. DOI: 10.1111/dsu.12149. arXiv: arXiv:1011.1669v3. URL: http://dl.acm.org/citation.cfm?id=324538.

[45] B. Neumayr and M. Schrefl. "Abstract vs concrete clabjects in Dual Deep Instantiation". In: *CEUR Workshop Proceedings* 1286 (2014), pp. 3–12. ISSN: 16130073.

[46] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. "Empirical assessment of MDE in industry". In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. 2011, p. 471. ISBN: 9781450304450. DOI: 10.1145/1985793.1985858. URL: http://portal.acm.org/citation.cfm?doid=1985793.1985858.

[47] D. Kaslow, L. Anderson, S. Asundi, B. Ayres, C. Iwata, B. Shiotani, and R. Thompson. "Developing a CubeSat Model-Based System Engineering (MBSE) Reference Model - Interim status". In: 2015-June (2015). ISSN: 1095323X. DOI: 10.1109/AERO.2015.7118965.

[48] K. Vipavetz, T. A. Shull, and J. Price. *Interface Management for a NASA Flight Project using Model-Based Systems Engineering (MBSE)*. Is. INCOSE International Symposium, 2016, pp. 1–15.

[49] C. Atkinson and R. Gerbig. "Melanie: multi-level modeling and ontology engineering environment". In: *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards* Lml (2012), pp. 5–6. DOI: 10.1145/2448076.2448083. URL: http://dl.acm.org/citation.cfm?id=2448083.

[50] J. de Lara and E. Guerra. "Deep Meta-modelling with META DEPTH". In: *Objects, Models, Components, Patterns: 48th International Conference, TOOLS In: Lecture Notes in Computer Science*. Vol. 6141. Springer, 2010, pp. 1–20. ISBN: 9783642139536. DOI: 10.1007/978-3-642-13953-6_1.

[51] R. Gerbig, C. Atkinson, J. De Lara, and E. Guerra. "A feature-based comparison of melanee and metaDepth". In: *CEUR Workshop Proceedings* 1722 (2016), pp. 25–34. ISSN: 16130073.

[52] B. Volz, M. Zeising, and S. Jablonski. "OMME - A Flexible Modeling Environment". In: *ICSE 2011 Workshop on Flexible Modeling Tools* (2011).

[53] M. Perrotin, E. Conquet, J. Delange, and A. Schiele. "TASTE : A Real-Time Software Engineering Tool-Chain Overview , Status , and Future". In: *SDL 2011: Integrating System and Software Modeling*. Springer Verlag, 2012, pp. 26–37.

[54] P. M. Fischer, M. Deshmukh, V. Maiwald, D. Quantius, A. M. Gomez, and A. Gerndt. "Conceptual data model: A foundation for successful concurrent engineering". In: *Concurrent Engineering* (2017), p. 1063293X1773459. ISSN: 1063-293X. DOI: 10.1177/1063293X17734592. URL: http://journals.sagepub.com/doi/10.1177/1063293X17734592.

[55] T. Kühne. "A story of levels". In: *Models 2018*. Copenhagen, 2018. URL: https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/10/multi_paper6.pdf.

[56] S. Lauesen. "Functional requirement styles". In: *Software Requirements*. Addison-Wesley, 2002, pp. 71–153. ISBN: 0 201 74570 4.

[57] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008, p. 744. ISBN: 0-321-33188-5.

[58] A. Kovalov, E. Lobe, A. Gerndt, and L Daniel. "Task-Node Mapping in an Arbitrary Computer Network using SMT Solver". In: *Lecture Notes in Computer Science*. Ed. by N. Polikarpova and S. Schneider. Vol. 10510. Springer, Cham, 2017, pp. 177–191. ISBN: 9783319668451. DOI: https://doi.org/10.1007/978-3-319-66845-1_12.

[59] A. Lange and C. Atkinson. "Multi-level modeling with MELANEE". In: *Models 2018*. 2018. URL: https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/10/multi_paper3.pdf.

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig