Noname manuscript No.
(will be inserted by the editor)

Utilizing Multi-Level Concepts for Multi-Phase Modeling

Context-Awareness and Process-Based Constraints to Enable Model Evolution

Tobias Franz · Christoph Seidl · Philipp M. Fischer · Andreas Gerndt

Received: date / Accepted: date

Abstract In model-based systems engineering projects, engineers from multiple domains collaborate by establishing a common system model. Multi-level modeling is a technique that can be used to model the development from abstract ideas to concrete implementations. However, current multi-level modeling approaches are not adequate for processes with multiple modeling phases that might have to be rearranged later. In this paper, we introduce *multi-phase modeling* that utilizes concepts of multi-level modeling by considering a description of the expected phase ordering per domain. Constraints aware of this context can express that certain elements are only valid in specific phases without having to determine a concrete phase ordering for a particular model. This enables using multi-phase modeling in flexible workflows, adapting to changing requirements and the definition of access rules in domain notation. We show feasibility of this multi-phase modeling by applying it to multiple real-life systems engineering projects of the aerospace domain.

Keywords Model-based Systems Engineering · Multi-level Modeling · Domain-Specific Languages · Systems Engineering

T. Franz
E-mail: tobias.franz@dlr.de

T. Franz · P. M. Fischer · A. Gerndt
German Aerospace Center (DLR), Institute for Software Technology, Braunschweig, Germany

C. Seidl
IT University of Copenhagen, Department for Computer Science, Copenhagen, Denmark

1 Introduction

Creating a spacecraft requires different engineering domains working together. This collaboration is backed by a systems engineering process, defining project phases, milestones and deliverables [2]. These project phases cover aspects such as the design, assembly, integration, operation and finally the disposal of the system. Along these phases, information has to be shared across involved domains. Classical systems engineering handles this information exchange by manually produced documents. Model-based systems engineering improves this information exchange by data models [35]. Most of these data models are inspired by classical modeling approaches, such as SysML and UML. In early phases, engineers start modeling coarse and abstract models of the system. In later phases, the system models get reused and refined [18].

Figure 1 illustrates a corresponding engineering use-case. It presents a model of camera-based navigation systems. The displayed system consist of cameras and computers. The engineering process of this use-case follows the order of Definition, Configuration and Assembly, as often applied in the space domain. During design definition, engineers collect and model required components. These components, such as a Small-Camera, HD-Camera or UV-Camera, are later used in the final products. In the next step, these components are configured into first product templates, such as a budget version including a Small-Camera. Finally, configurations are assembled to reflect the real-life products. Along this process, different properties of the system are modeled. For example, at design definition, the resolution of different camera types is specified. Other properties such as the position are composition-related. They can only be modeled in the configuration or later. It cannot be

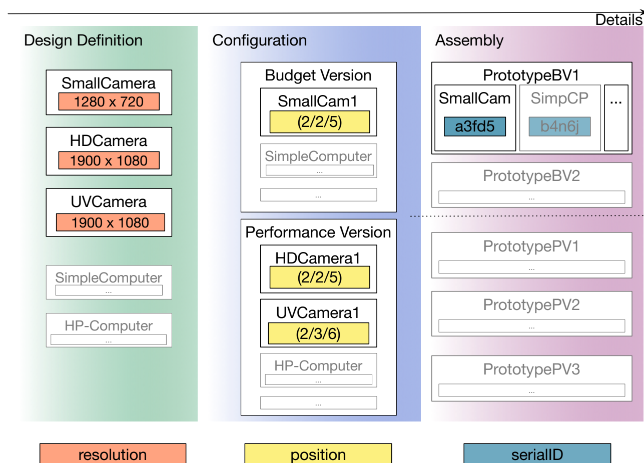


Fig. 1 Example engineering use-case depicting a model-based development process with three steps. Each step is based on previous information and adds more details to the model.

specified during the first definition phase because an abstract camera does not yet have a position. This simple example indicates that modeled information depends on a process. The resolution should not be edited after the definition, a position in this first phase does not exist.

Implementing the above model in classic SysML/ UML leads to various problems. Modeling the camera as a class with attributes for position and resolution has two drawbacks. First, it can only be instantiated once, e.g. in the design definition step. As a consequence, it cannot be instantiated in the configuration or assembly step anymore. Second, the position property is already present during design definition which is semantically incorrect. As mentioned, an isolated element in the definition cannot have a position.

Multi-level modeling solves some of these issues. In its classic form, it is based on elements that combine class and object facets and thereby allow instantiation in multiple classification levels [7]. In the previous example, this enables to first instantiate the camera in the design phase and, then, to re-instantiate it in the configuration and assembly levels. Some multi-level modeling approaches, such as M-Objects, allow specifying in which level a value can be set [31]. That allows to specify, e.g., that the resolution property has to be modeled in the configuration level.

Recent work by Guerra and de Lara argues that we need more flexibility in modeling and they propose a modeling process with configurable flexibility for the different process phases [21]. They propose an architecture which is based on an explicitly modeled process model.

While our use-case requires a similar amount of flexibility for the different phases, a process model can help to model the process-dependent information of Figure 1. To specify which information is modeled in which process phase, we need a mapping of system aspects to the phase when these have to be modeled.

To achieve this, this paper defines an engineering modeling paradigm based on a new context model. It indicates what system details are intended to be modeled at which modeling phase. It captures the modeling process and enables constraints, such as that a position is only available after a specific phase. Every set of new types added to the model is accompanied with such a context model. This approach offers several advantages:

1. It allows to specifically define a modeling process for new model elements. This modeling process can be adjusted to domain specific semantics and needs.
2. The order of modeling phases is defined by the individual context model. Adding new phases to the context model introduces new layers in the system model.
3. The context model is the basis for constraining properties. Their validity can be customized to flexible ranges of modeling phases. This way, the order and content of elements is based on semantic context rather than numbers.

The context model represents the development process and contains its phases as elements. For the previous use-case, the context model contains elements for `Definition`, `Configuration` and `Assembly`. Constraint-wise, it allows instantiating the resolution in the definition phase and the position in the configuration phase. Altogether this approach enables an effective use of multi-level modeling concepts in systems engineering. It is based on the initial idea of context-aware potencies, but offers a fundamental addition by the context model.

This paper is structured as follows: Section 2 introduces relevant literature in the field of systems engineering and multi-level modeling. Section 3 highlights the problems of these techniques and poses requirements for an application of multi-phase modeling for systems engineering. Section 4 presents a new engineering modeling paradigm for development process with multiple modeling phases. Section 5 evaluates these concepts by applying them to scenarios of interdisciplinary projects for model-based space system development. Finally, Section 6 discusses related work and compares our methodology with similar concepts.

2 Background

Model-Based Systems Engineering (MBSE) targets substituting classic system engineering documents with models [35]. Introduced models may be used for specifications and analysis of systems. Collecting system information in a centralized model also improves communication. Such a central model provides clear rules for information exchange across all stakeholders.

Systems engineering usually follows certain phases. For space systems, these phases are known as *lifecycle phases* and are defined in standards by the European Cooperations for Space Standardization (ECSS)[2]. The lifecycle starts with `Phase 0` and is followed by A, B, C, D, E and F. Each phase describes tasks, deliverables and design reviews that have to be fulfilled in the phase. However, project goals and derived requirements in these phases are subject to frequent changes [20].

Paige et al. describe that models have to be updated along the development process [34]. They distinguish between two different model evolution strategies: One where the underlying metamodels change, the other where theses metamodels remain without changes. In implementations whereby metamodels evolve, corresponding system models have to be transformed to comply to newer versions of the metamodel. According to Galvão and Goknil, system models thereby transform from abstract system descriptions to models with more concrete details [18]. Every evolution step has its own metamodel and modeling language. Thus, modeled elements in later system models cannot be implicitly traced back to their previous models. Such traces have to be modeled explicitly. Galvão and Goknil present a mechanism to automatically model explicit traces within the system model transformation [18]. In contrast, in implementations where metamodels do not evolve, metamodels need to anticipate future needs of the system model [34]. Fischer et al. present a data model that is used for several project phases of space system development projects [15]. In their approach, the metamodel provides generic modeling elements. Types for these generic elements can be created later. This mechanism is based on the Type Object and the dynamic template pattern [24][29]. These pattern introduce elements to decouple classes from their instance [29]. Implementations thereby contain metamodel types for type definitions and their instances (e.g. *TypeDefinition* and *TypeInstance*) [29]. Atkinson and Kühne argue that the mismatch between a problem and its technical implementation leads to accidental complexity [10]. To reduce accidental complexity, they introduced a new modeling paradigm, *multi-level modeling* (MLM).

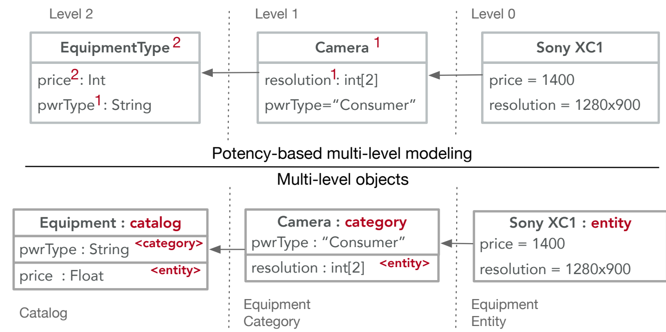


Fig. 2 Multi-level modeling with Potency (top) and M-Objects (bottom).

Multi-level modeling is based on the concept of a new element, called clabject [10]. It combines aspects of **cla**ss and o**bject**. Assuming the facet of a class, a clabject can be instantiated to a clabject with the facet of an object. Due to its nature as clabject, this element can assume the facet of a class again. Therefore, it can be instantiated several times, thus, introducing multiple levels of instantiation [7].

This kind of type-instance relation differs from the classic language-defining linguistic typing dimension [8]. The classic modeling dimension was introduced by the Object Management Group (OMG) to support metamodeling [1]. Their four-level hierarchy starts with a metametamodel, named M3 or Meta-Object Facility (MOF). The M3 model defines a basic language to create metamodels in the M2 level. A prominent example of a M2 model is the UML metamodel. M2 models describe the elements of M1 models. In case of the UML metamodel, M1 models are written in UML. Models in the M1 layer represent elements from the real world (M0). In contrast to this dimension, Atkinson and Kühne call their new typing dimension ontological dimension [8]. The ontological type-relation describes a classification within one linguistic level. To express that ontological levels are usually within one linguistic level, Atkinson and Kühne introduced the term Orthogonal Classification Architecture [9].

Potency is a constraint mechanism to control the instantiation depth of elements in multi-level modeling [7]. As shown in the upper part of Figure 2, it is a positive numeric value. Potencies can be assigned to clabjects as well as their properties. On properties, it describes at which level a property is instantiated and thus values can be assigned [10]. The property `price` on the `Equipment` can be instantiated until Level 0; the `pwrType` only until Level 1. Additionally, in recent work, Kühne discussed a mechanism called *order-alignment schemes* [26]. With this principle it is possible to shift a classification ensemble up and down in

the level hierarchy [26]. Macías et al. presented another way of specifying potencies by allowing to set a *min* and *max* potency value, thus, enabling ranges as potency value [30].

Fischer applied multi-level modeling techniques to an example of systems engineering [14]. He highlights the potential of multi-level modeling for systems engineering. Thereby, he shows several levels of instantiation and that some properties of objects are only semantically valid in certain stages of the modeling process. He argues that potency based on static numbers introduces limitations in case the process needs to be adjusted [14].

Besides potency, there exist further concepts how types and their properties can be bound to specific levels: Neumaryr proposes a mechanism called *multi-level objects* (M-Objects), which are not assigned to an explicit level but allow the direct concretization of objects on different levels [32]. As shown in the bottom part of Figure 2, M-Objects' properties explicitly target a level when they have to be instantiated.

Literature also shows how multi-level modeling can be integrated into practical modeling processes: Hinkel proposed an approach where deep modeling can be implemented with the existing EMOF standard, thus, enabling to use existing tools and transformation mechanisms with multi-level modeling [22]. Guerra and de Lara argue that modeling needs more flexibility and propose a configurable architecture that allows different levels of flexibility within the modeling process [21]. Their idea is based on an explicit description of the modeling process and thereby allows different levels of strictness within this process. Atkinson and Kühne also discussed modeling process and introduced *metamodeling spaces* to recognize that it is hard to integrate different modeling domains into a single modeling hierarchy [6]. Neumayr et al. presented *dual deep modeling*, which enables integration of models with different clabject hierarchies into a global consistent representation [33]. Dual deep modeling differentiates between source and target potency, where source potency corresponds to the level in the own clabject hierarchy whereas the target potency specifies the level in the referenced hierarchy.

3 Modeling of Phase-Specific Information

Interdisciplinary systems engineering involves several different domains working on a central system model. As shown in Figure 3, information in models is re-used in consecutive process phases and follow-up projects. Thus, information modeled in early project phases is foundation for models of later phases. However, in the process, new projects, system variants, configurations and prototypes splinter from the common base model. This leads to divergence of the models in later phases of projects. Follow-up projects might e.g. focus on specific aspects of a system or integrate ideas/solutions into another system.

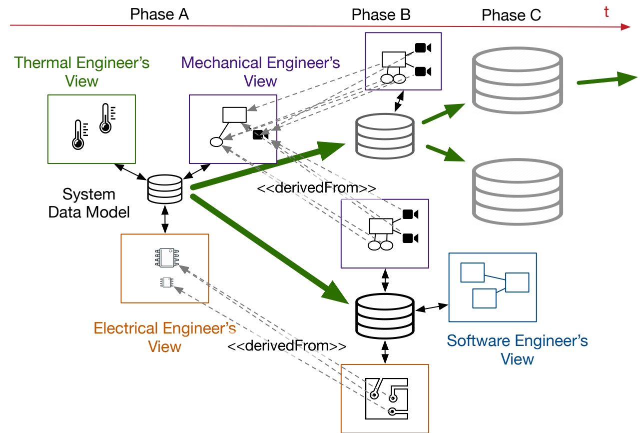


Fig. 3 Modeling in an environment with multiple process phases. Consecutive project phases and follow-up projects use the results from previous models.

Multi-level modeling can be used to model the relation between abstract concepts in the beginning of the project and concrete solutions in later phases. However, with models diverging from each other in the modeling process, the same base types have to be used in different project contexts. Figure 4 shows two different modeling contexts with an equipment. In the upper context, a first derived element is a `UVCamera`. In the lower context, the first derived element is a camera and the `UVCamera` is the third element. Both modeling contexts are derived from a common initial phase, creating the equipment. Different project requirements, then, led to different levels of detail when modeling the camera aspects.

(1) We define a process in which domain aspects are modeled in multiple consecutive development phases as *multi-phase modeling*.

Although elements in different phases are modeled from different abstraction levels, their relation is not necessarily the non-transitive classification relation. This highlights a key difference between our use-case and classic multi-level modeling: systems engineering requires model 'levels' that are not classification levels. Thus, in contrast to levels, phases have to allow other inter-phase relationships between elements than classification. To reflect this, we call the relation of elements between different phases *derived from*. Multi-level modeling was not designed for multiple modeling phases and, thus, does not support re-purposing elements based on dif-

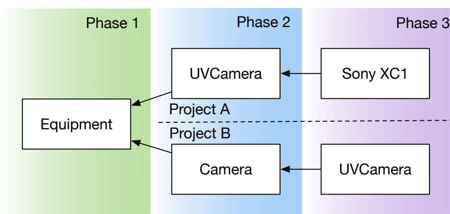


Fig. 4 Equipment in different application contexts. Depending on the project's required level of detail, different modeling phases are necessary. As shown by the `UVCamera`, phases are not necessarily classification levels.

ferent contexts. However, adding a dedicated modeling phase for the `Camera` in all contexts would introduce modeling overhead. Some projects require modeling special camera attributes, others not. It is context-dependent in how many modeling phases the camera needs to be present. This poses the challenge how to target the `UVCamera` with modeling constraints (in the sense of e.g. potency). As argued in our initial example in Figure 1, engineering data are phase-specific. Indeed, engineering processes usually clearly define what has to be modeled in which process phase. For example, the configuration of the camera has to be done before it is assembled, thus, the resolution must not be changed in later project phases. This needs to be reflected in a modeling environment. There needs to be some way to specify what has to be modeled, e.g., in the modeling phase of the `UVCamera` in Figure 4. As shown in the figure, however, it depends on the project context in which phase the `UVCamera` is modeled.

3.1 Requirements for Multiple Modeling Phases

Modeling constraints can be used to reflect a systems engineering process by specifying that, e.g., the camera's resolution cannot be created after a specific phase. However, constraints with references to, e.g., static numbers need to be adjusted if the process changes. As types and corresponding constraints might be created in a common base model and are then reused in different specific project contexts, changing the common base model is impossible. Thus, model elements and their modeling constraints need to be reusable in models with different processes.

Furthermore, constraints need to reflect various different process requirements: In our example (Figure 1), the camera's `serialID` can only be specified starting from the assembly phase on. Thus, to prevent misunderstandings, values must not be set before. Moreover, a position of an element in a bare collection of types does not make sense. Thus, the position property in a definition phase is not valid. When defining properties, it needs to be possible to specify in which phases these can be used.

In systems engineering, it is necessary to incorporate models from different domains into one system model. These models from various domains will have different phases, so some kind of integration / synchronization mechanism is necessary.

If model elements are implemented similar to clabjects in multi-level modeling, thus having a class and object facet, it is possible to add new properties to their 'instances'. However, in interdisciplinary systems engineering it is problematic if everyone can add their own properties to the model. The same property might be created multiple times, potentially with different names and not aligned within the project team. To solve this problem, creation of new properties needs to be restricted to specific modeling phases.

Thus, to facilitate modeling in interdisciplinary engineering environments with multiple modeling phases, we pose the following requirements:

R.1 Model elements have to be able to be re-used in contexts with different modeling phases.
R.2 Accessibility of individual elements has to be configurable to flexible ranges of phases.
R.3 Models have to allow integration of elements from different domains (originated in different models).
R.4 The modeling environment has to constrain type extensions (e.g. definition of new properties).

4 Context-Aware Multi-Phase Modeling

This paper introduces an engineering modeling paradigm that enables to model interdisciplinary systems throughout multiple project phases. To represent different abstraction levels of these phases, we utilize multi-level modeling concepts. However, as classic multi-level modeling is not explicitly designed for multiple modeling phases, we extend and customize its concepts. As developed in Section 3, systems engineering requires modeling concepts to be customized to different application contexts. A specification of what has to be modeled in which development phase provides a description of such context. Therefore, this section introduces a *context model* that describes the modeling process of an application. It outlines the modeling phases and their order in the corresponding application context.

(2) We define the *context model* as a formal description of the phases a set of domain elements passes through along a development process.

An instance of a domain element in a particular modeling phase is automatically in a state defined by the

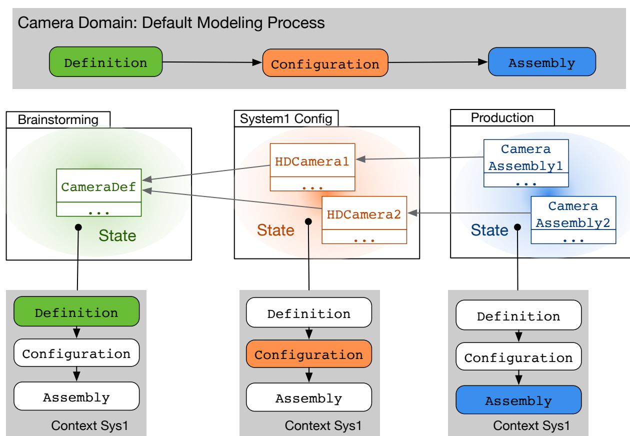


Fig. 5 Multi-phase modeling process for the camera example project. The camera is derived to a repository of the configuration phase and finally to multiple production prototypes.

respective phase in the context model. If no corresponding phase is defined in the context model, creation of the domain element is not possible in the current modeling phase.

4.1 Modeling Phases in a Development Process

In systems engineering, the development process and organizational structures capture how experts from multiple domains contribute to the development of a system model. The development process has direct influence on how a model is created, edited and used. Successive development phases derive output developed in previous phases and add further details. Thus, in a broader sense, evolution of projects resembles instantiation over multiple phases. As a consequence, context models in systems engineering resemble process models. A domain expert will probably not be able to answer the question of an instantiation depth of a domain element, whereas it is more likely to be successful to ask when in a development process it can be edited. Thus, a context model maps the development process to a multi-phase system model. It corresponds to a plan in which modeling phase the different aspects have to be added.

As shown in Figure 5, elements are derived to multiple modeling phases. In our running navigation system project example, as part of a first brainstorming session, relevant concepts are defined. The camera concept is derived two times, in a system configuration and finally again in multiple assembly prototypes. Elements in these phases can belong to different abstraction levels: the camera type (`CameraDef`) is classifier for the two `HDCamera` instances. It would be possible to model this example in a three level multi-level model that would look similar to Figure 5. The `Brainstorming` modeling phase corresponds to a model level 2, the `System1 Config` resembles level 1 and finally the `Production` phase represents level 0. However, our example is based on a modeling process. In the initial definition phase, we might not yet know if the project is continued and which of the concepts are going to be used. Maybe the finalized system is actually reused and further customized in a follow-up project. We know, however, that our development process requires certain properties to be handled at specific phases in the process: To pass an early milestone we have to successfully run specific simulations, that require a system configuration to be done. Furthermore, we can only assemble the system, if we know the specific serial ids of the system parts. To incorporate this information and to handle corresponding model evolution, multi-phase modeling is designed around an explicit process description, which can be seen in Figure 5. Each process phase is represented in a context model, and the domain elements in a repository always belong to one of these phases.

4.2 Context Description as Guide for Model Evolution

To reflect that results from previous project phases are re-used, enriched with details and applied in consecutive steps, `Phases` are inspired by levels in multi-level modeling. Elements can have instances in each of these phases, which are represented in the context model. Initially, elements are in the first phase of a context model; deriving them to a new modeling phase changes their *phase state* to the next phase in the context model. The `CameraDef` domain element in the `Brainstorming` model is automatically in the `Definition` phase, as defined in the context model. Recreating the camera in the `System1 Config` model, automatically changes its state to be in the `Configuration` phase. In contrast to levels in multi-level modeling, the phase state (an element's level) is not derived from the container of the element (the level in MLM), but from a context model. If the derivation depth of an element is two, then its state is in the second phase of the context model, independently of its container.

4.2.1 Structure of Multi-Phase Modeling

Figure 6 shows an excerpt of the metamodel for multi-phase modeling. Central types are:

– *Repositories* are the root containers and have a specific modeling goal, such as ”Model the configuration of a satellite” or ”Create domain elements for the thermal domain”. They combine required modeling aspects from other repositories if needed.

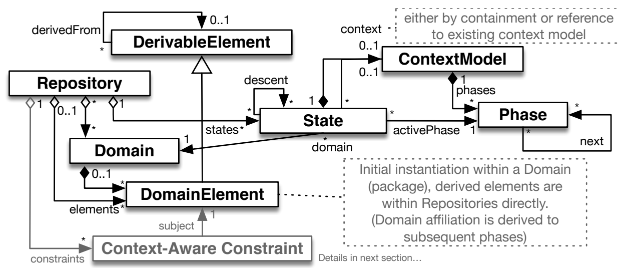


Fig. 6 Metamodel illustration for context-aware multi-phase modeling. Repositories contain domain elements and their corresponding 'level' state. This state is derived from the active phase in the state's context model. (Constraints are explained in Section 4.3).

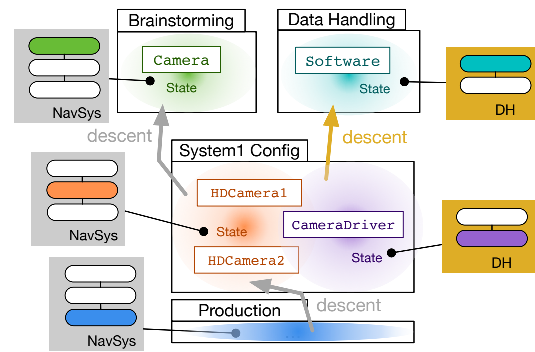


Fig. 7 Modeling with multiple domains and their context models. Repositories can combine domain elements from multiple other repositories. The phase state of the domain elements is evaluated locally with their domain's context model. A global phase/level for different domains does not exist.

– *Domain elements* are used to model this specific goal. They are implemented similar to clabjects in classic multi-level modeling and, thus, as clabjects can be instantiated in multiple levels, domain elements can be added into multiple phases. This is archived by the `DerivableElement` whose instances can be derived from other instances, thus, creating derivation chains.
– *Domain*: Initial creation of domain elements is done within a package element, called domain. Derived domain elements, optionally from different domains, are added into repositories directly. The domain affiliation (containing package of the initial element) is passed on to all derived elements. Thus, the domain of a domain element can be determined by the first element in the derivation chain. Domain elements of the same domain share a common context model.
– The *Context Model* is a directed graph with modeling *Phases* as vertices. Each phase represents a state of corresponding domain elements in their life-cycle. This means a context model describes in how many modeling phases a domain element can be represented. Compared to multi-level modeling, phases are similar to levels, however a phase is not a (conceptual) container within a model, but it is a (life-cycle) state of corresponding domain elements. Repositories contain and manage these states.
– A *State* is automatically created once a domain element is added into a repository. All of a domain's elements in a repository have to be in the same phase. To specify this *phase state*, the `State` contains a context model and a link to the currently active phase. If an existing context model is reused, the context model can also be referenced. The `descent` relation links to the previous state space and thereby specifies which elements of a domain can be derived into this space (in form of a repository). The combination of a repository and a contained state corresponds to a level in multi-level modeling.

Repositories are bare structural containers and do not influence the domain element's phase state. Instead, the domain element 'emits' its state to the repository. If one domain element in a repository is in, e.g., the configuration phase, then all elements of this domain have to be in this state. To achieve this, the state links to the domain, not to the domain element: it influences all elements of a domain. After the camera concept was added into the `System1 Config` repository, all further domain elements in this repository are automatically in the configuration phase. Then, domain elements with a different derivation depth cannot be added into such a repository.

As shown in Figure 7, elements of different domains in the same repository can be in a state defined by different context models. Thus, they are in different phases. Elements of different domains in the same repository do not even need to have a common phase. By using context models, phases are broken down to domains. Repositories themselves do not have any conceptual order. Ordered are the domain elements within the repository, as specified by the context model. The `System1Config` repository in Figure 7 combines domain elements from other repositories with elements from different domains. The `HDCamera1` and the `Camera Driver` are in the same repository, but in different states, defined by different context models. In the example, the data handling aspects are relevant for the system configuration; the production repository, again, contains only domain elements from our navigation system.

4.2.2 Context Model Notation

Contexts are modeled by describing the phases of the surrounding modeling process. This description can be an ordered list of phase identifiers or a more complex directed graph. As shown in Figure 6, the phase's prop-

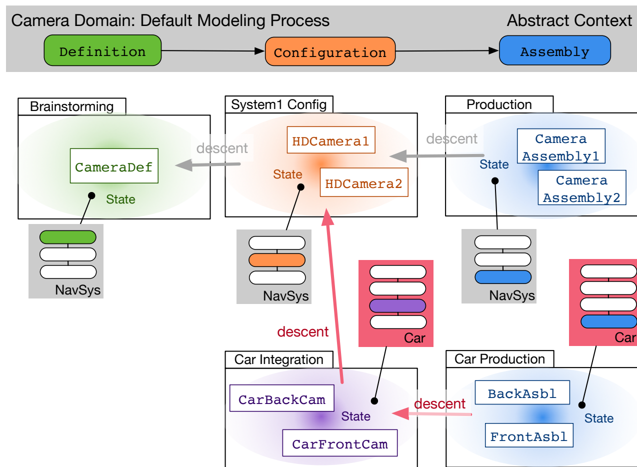


Fig. 8 Extension of Figure 5: adds a project that customizes the context model so that the domain elements can be derived to an additional modeling phase for the integration of the camera elements into cars. The red context models are customized.

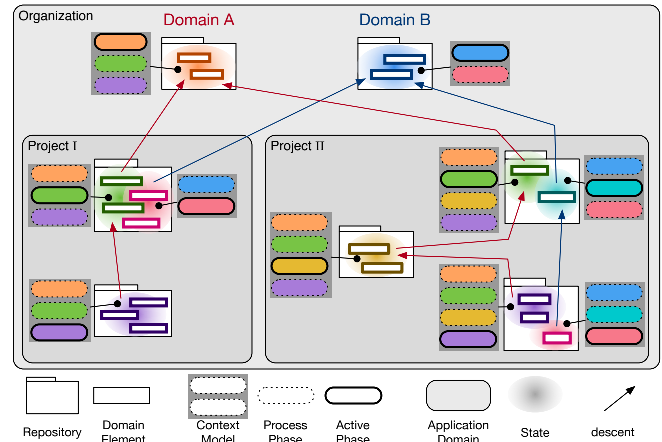


Fig. 9 Flexible structure of multi-phase modeling with elements from different domains. The context model of Domain A is shown left to the Repository, the context model of Domain B is on the right side.

erty `next` allows multiple next phases and, thus, multiple paths in the context model. If a domain element is instantiated into a new stateless repository and the domain element's state has multiple options for the next phase, then the modeler can select which of these paths in the context model is followed. The directed graph can also contain circles, enabling iterative processes and effectively allowing an infinite derivation depth of domain elements.

4.2.3 Customization of Context Models

One of the motivations for context models was to achive more flexibility where needed by keeping control where necessary. In this section we discuss how this flexibility can be utilized by customizing context models. Continuing the camera example of Figure 1, the basic context model contains three elements for the phases of definition, configuration and assembly. A customized context with an additional integration phase allows elements to be in a repository representing the integration phase. In our example, the same camera definition is used in two different modeling projects, with different development processes.

Figure 8 shows how this project-specific customization of contexts works. The initial context model describes a common process in the navigation system developers' company. This common process contains phases for the already known order of definition, configuration and assembly. It can be described in an *abstract context model*, without any repository or domain elements. (Top of Figure 8) A project that follows this generic process re-uses the context model implicitly when instantiating a domain element into a repository. A different project that targets to integrate this navigation system into other systems, such as cars, requires an additional integration phase. Thus, a more complex project can customize the existing common context model and derive its extended context model. Domain elements in the project's system model then contain an additional derived form of the camera in a fourth modeling phase. This extended project can then continue to use the system data from the first two phases without having to modify them.

To ensure consistency, context customization has to follow some rules:

1. No phases can be deleted
2. Phases can be added after the currently active phase
3. Phases can be edited (name, link to next phase) after the currently active phase

Since the domain elements in the referenced repositories cannot be changed, only phases after the current phase in the context model can be changed (add phase / change path). This reflects that we cannot modify the past but only future modeling phases. As context models resemble the modeling process, customization of contexts is usually done if the process changes or if the model is used in projects or organizations with specific requirements.

4.2.4 Context Models with Multiple Domains

Figure 9 visualizes the structure of multi-phase modeling with multiple context models and projects. A *Context scope* consists of a repository and its corresponding context models. Every domain element in a repository

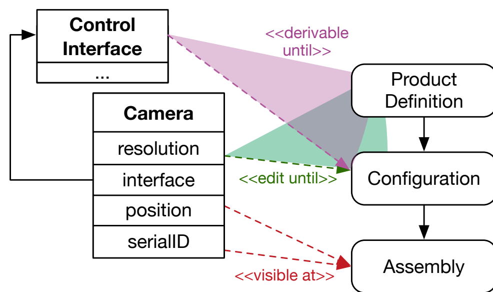


Fig. 10 Context-aware model manipulation constraints. Context models can be used to specify to which phases an element can be added to, when it is visible and can be edited.

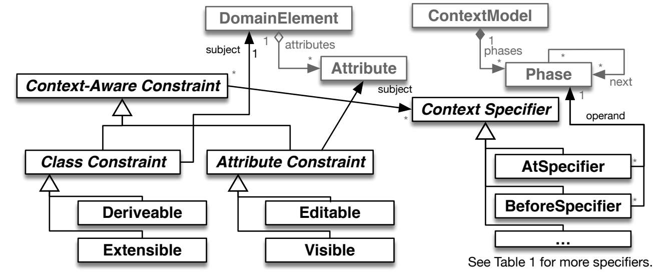


Fig. 11 Metamodel illustration showing relevant elements for context-aware constraints. There are constraint types that target DerivableElements and others that target its attributes. Phase specifiers link into the context model to select a range of modeling phases.

needs to have a phase specification in a context model within this context scope. Repositories do not follow a strict order, they are free floating. Ordering structure is the context model. Repositories can be used to define new elements or to connect these e.g. in a system model (which is then also contained in a repository).

The example shows an organization that models aspects of two domains, Domain A and Domain B. Each domain defines a set of generic domain elements and their corresponding context model. Projects derive and customize these domain elements in their system model. In the example, Project I sticks to the common context model. Project II customizes the context models so that elements of both domains are represented in an additional phase. For Domain A, this new modeling phase is added between the existing ones. Its domain elements for this additional phase are contained in a separate repository. This makes it possible to *synchronize* phases of different domains. The last repository of Project II contains elements derived from different other repositories. There is no strict order of repositories. Instead, domain elements are part of a conceptual `typing dimension` specific to their domain and defined by the context models.

4.3 Context-Aware Constraints

The context model enables to specify in which modeling phases the domain elements can be created. However, the main motivation for multi-phase modeling is to model phase-specific information. Depending on the phase, domain elements focus on different modeling aspects. In an initial brainstorming session for elements needed in a system, it does not yet make sense to model the position of these elements as a system does not yet exist. The information in which modeling phase the elements of the current repository are, is derived from the context model. We define context-awareness as knowledge of the context model and the state specifying the currently active context phase. In our camera example (Figure 10), after domain experts determined a specific resolution, it must not be changed in consecutive phases. Production workers should not be able to change the resolution as corresponding prototypes might be used to evaluate this property. Context-awareness enables to specify at which point in the modeling process which system aspects have to be modeled.

(3) We define *context-aware constraints* as a formal description of which constituents of a domain element are accessible in which modeling phase.

Thereby, a context-aware constraint allows customizing during which development phases selected properties of a domain element are visible, mutable, and/or assignable as well as whether the domain element may be derived or extended.

4.3.1 Structure of Context-Aware Constraints

Figure 11 shows the metamodel for context-aware constraints. Constraints always have a subject, a constraining type and a context specifier. The subject is a reference to a `DerivableElement` or one of its attributes, specifying which element is affected. The constraint type can be one of the following:

- `derivable`: Customizes in which phases the DerivableElement can be created in. Without any constraint, DerviableElements can be created in all phases as specified in the context model.
- `extensible`: Defines in which phases new attributes can be added to DerviableElements. Without constraint, it is not possible to add new attributes in any phase.
- `visible`: Specifies in which phases an attribute / a reference is visible. Without constraint, attributes are visible in all phases.

Table 1 Specifiers for context-aware model manipulation constraints.

Subject	Type	Phase Specifier
DerivableElement	derivable	always
	extensible	never; $x = \{\}$
Attribute /	visible	at A; $x = A$
Reference	editable	after A; $x > A$
		before A; $x < A$
		until A; $x <= A$
		from A; $x >= A$
		between A and B; $A < x < B$

- `editable`: Specifies in which phases an attribute / a reference can be assigned a value. Without constraint, attributes are editable in all phases.

If required additional constraint types can be added. The phase specifier allows to define when this constraining characteristic is active in the modeling process. To mitigate limitations of a fixed number of phases, constraints use references to the context model rather than hard-coded numbers (as derivation depth). Decoupling, achieved by these references to the context model, does not predetermine an order of phases and thus allows changing it later. It enables precise constraints by leaving room for customization of the modeling process. Model elements can be targeted by multiple different constraints. This way, it is possible to, e.g., configure if an element is visible and editable independently from each other by using two separate constraints. Furthermore, constraints are inherited to child types (both, through inheritance in the same phase and also derived elements in later phases), but can be overwritten, similar to attributes in classic object orientation. Constraints are bound to their subject and share their life-cycle.

4.3.2 Constraint Specification

The graphical notation for constraints, applied in Figure 10, uses a link between a domain element and a phase in the context model. Start point of the link is the constraint's subject; end point the context phase in which it is active. The color represents the constraining type. As textual representation of the `edit until` constrain in Figure 10 we recommend a structure like this:

`<Subject>`	`<Constraint Type>`	`<Phase Specifier>`
`resolution`	`edit`	`until Configuration`

The different constraint types (collected in Table 1) use specifiers for ranges of phases in which they are valid. For example, according to the `edit until` constraint, the camera's resolution can be edited until the configuration phase. With a context model as shown in Figure 10, this means it can be edited in the first two phases. Its position and serialID property are visible in the assembly phase. The `derivable until` constraint specifies that the `Control Interface` can be added to the first phases. This kind of constraint corresponds to potency in multi-level modeling. It allows specifying that elements are applicable to not all of the phases introduced by its domain's context model. To fulfill Requirement 4, new properties cannot be added to domain elements in all modeling phases but only in these where it is explicitly allowed.

To increase flexibility of editing rules, the constraint mechanism supports targeting ranges of modeling phases. This enables to specify that properties are, e.g., editable after or between specific phases. Figure 12 shows how this mapping of properties to domain elements works. The figure contains three constraints:

C1 `Camera.resolution edit until Configuration`
The camera's resolution can be seen in all phases of this domain element, however, it can only be edited until the element is in the configuration phase. (`until`: configuration phase included; `before` would not include it).

C2 `Camera.position visible after Configuration`
The camera's position property is visible after the configuration phase (`after`: configuration phase not included; `from` would include it). It is also editable because we do not have any edit constraint.

C3 `Camera.serialID visible at Assembly`
The camera's serialID property is visible in the assembly phase.

The first element of the camera will be in the `Product Definition` phase and according to constraint (C1) it contains the resolution property. A specification of a resolution in a phase before the (final editable) configuration phase is handled as default value. The element with a derivation depth of three is in a phase after the `Configuration` phase (C2) and at the `Assembly` phase (C3), so it also contains the position and serialID property. According to (C1) the resolution property can still be seen but not edited anymore.

When specifying constraints, it is only possible to reference already modeled context elements. `Range constraints`, however, provide a way to consider context customization. As the camera's position in Figure 12 is visible after the configuration phase, it is also accessible in an additional integration phase. This makes the selection of phase specifiers from Table 1 more relevant. While the specifiers `after Configuration` and `from Assembly` result in the same outcome with the initial context model, an offspring in an additional

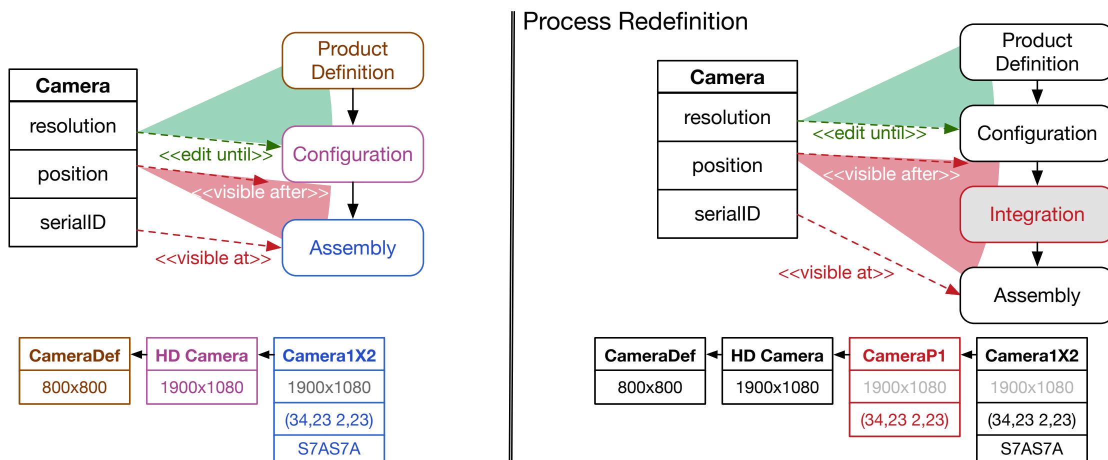


Fig. 12 Context-aware element creation and context customization. Depending on the constraints, properties are mapped to the elements. Range constraints can be used to consider process customization.

integration phase in between is accessible with the `after` keyword.

4.3.3 Composition of Constraints

As mentioned before, it is possible to address one element with multiple constraints and each constraint can have multiple phase specifiers. While this is necessary to, e.g., specify when elements are visible and editable independently from each other, it may also lead to conflicts. Constraints could specify that attributes are editable when not even visible. Constraints of attributes and their corresponding DerivableElement could be conflicting. To ensure consistent behavior, constraints are evaluated following the order: `derivable, extensible, visible, editable`. Phase specifiers are evaluated according to the order of their specification. So if an element is not visible but editable, it is simply not visible. Furthermore, if necessary in the modeling environment, constraint editors can support model users by highlighting inconsistencies and conflicts.

Summarizing, the combination of context models and context-aware constraints allow to specify when in the modeling process domain elements can be created, and their representation can be customized to model phase-specific information.

5 Systems Engineering Case Study

To evaluate multi-phase modeling against the requirements in Section 3, we apply it to examples of industrial relevance. In detail we analyze if it:

– fulfills the requirements from Section 3
– is possible to be integrated into systems engineering projects

Fischer presented a systems engineering scenario in which he showed challenges of potency-based multi-level modeling [14]. In his example, projects exist in which it is necessary to add an additional model 'level' because one system configuration might be used for the integration of different payloads. He models a telecommand, which represents a command for remote control. Aspects of this telecommand need to be modeled at different abstraction levels of the system. Furthermore, the telecommand has properties which are required but only known at specific modeling phases. Modeling this scenario requires customization of the modeling process as required by Requirement R.1.

Furthermore, to demonstrate the extension's practical applicability and to analyze its impact on the modeling process, we implemented a prototype and applied it to real world data. Therefore, we integrated multi-phase modeling into a model-based systems engineering tool for space system development. Besides the constraint mechanism for model elements, as required by Requirement R.2, we also evaluate how concepts from different domains can be integrated, as required by Requirement R3.

5.1 Handling of Changes in the Numbers of Phases

Figure 13 shows how, according to Fischer, a typical structure describing an attitude and orbit control system (AOCS) can be modeled using multi-level modeling [14]. Part of the AOCS domain is an element for telecommands that is used to specify commands for remote control of the system. Telecommands have properties, which are known in specific phases of the engineering process only. In this example, the telecommand is controlling a reaction wheel, which is a component of

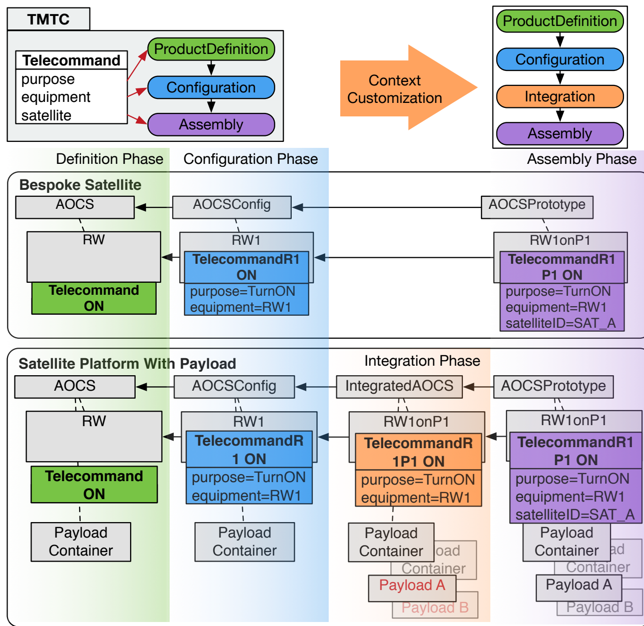


Fig. 13 Mapping of element properties to model levels. Overriding the context model allows adding new levels to a system model.

the AOCS subsystem. Its initial type definition in the system model is derived to different configurations of a satellite. These configured elements are then derived as assembly units in different prototypes per configuration. In the first level, as part of the definition of the reaction wheel, engineers specify that this element can be turned on with a telecommand. Each telecommand, however, needs information that is accessible in later levels only. Examples for such information are the satellite and equipment id. The scenario's challenge is the expectation that additional modeling phases have to be integrated between existing ones.

To model this use-case with multi-phase modeling, one has to create a context model first. The different levels, described by Fischer, are handled as modeling phases in our approach. Describing the development process in the context model allows mapping the telecommand properties to the different project phases. The purpose of a telecommand is only editable in the first definition modeling phase; the satelliteID in the assembly phase. With an additional phase, e.g. for the integration of different payload components, the satellite id needs to be edited in the fourth phase/level instead of the third. A customized context model for this development process solves this problem: the integration phase is added into the context model. As the property `satelliteId` was assigned to the assembly phase, the constraint remains valid also with an additional phase between the previously existing phases. It cannot be edited in the third integration phase. Thus, the main challenge of this scenario, handling of changes in the number of phases / model levels, is explicitly addressed and fulfilled by the introduction of the context model.

5.2 Interdisciplinary Modeling Environments

An important requirement for a systems engineering methodologies is to support composition of elements from different domains (Requirement R3). This section evaluates how multi-phase modeling handles this 'synchronization' of modeling phases. Figure 14 shows an example where elements from different domains are integrated into one system model. Domain elements are reused in several locations of a system. A camera concept is derived according to the modeling process of definition, configuration and assembly. It has two derived elements in a system configuration. A network port, modeled in another repository can be used to, e.g., connect the cameras and an onboard computer in the configuration repository. As shown in the figure, a specification of a `DSub2BusPort` with two pins can be instantiated in other repositories of the system model. Elements of this type, then, automatically contain the modeled pins and a reference to the type description. In this particular case, the 'derived from' relation between domain elements of different phases is actually an 'instance of' relation. Thus, multi-phase modeling can be used to model multi-level modeling aspects.

The input and output properties of the components represent a link of the structural domain to the networking domain. Besides the type of this relation, it is also possible to specify the phase of the referenced type: components only accept ports in the application phase as input and output (the `Port` type is rendered green, the application phase's color).

Context-aware constraints utilize a description of the modeling process to allow controlling how system elements are used. The port's property voltage is editable until the specification phase. This prevents users from changing it in the network domain's application phase. There, the property is still visible but read-only. Furthermore, according to the `derivable at` constraint, users cannot create domain instances of the network protocol in the application phase. The number of pins of a port is only editable in the specification phase, whereas the actual connection of these pins is done in the application phase. `Camera1` uses `Pin1` to connect to the onboard computer; `Camera2` uses `Pin2`. This way, context-aware constrains make it possible to distinguish between the different facets of the network elements. In the specification phase, model users can edit the aspects of the network specification (Port name, pins, voltage, protocol); in the application phase, users can

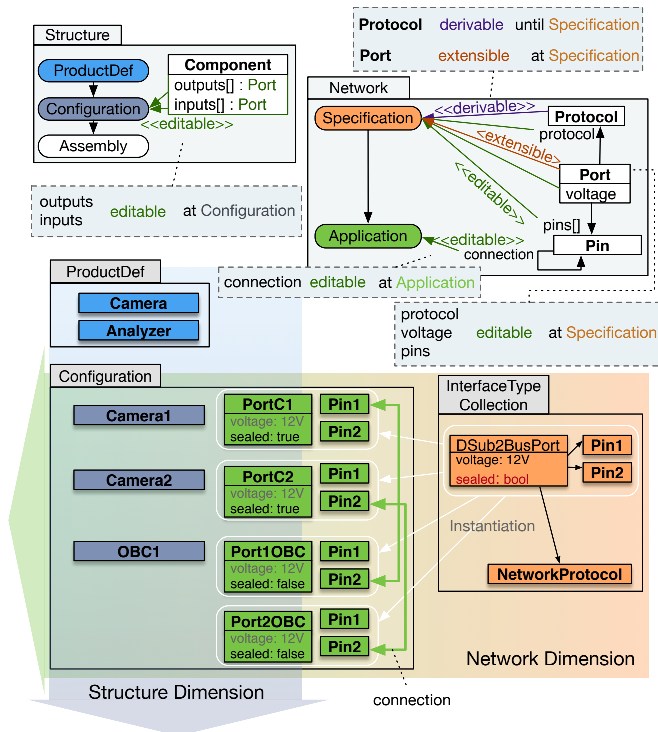


Fig. 14 Reuse of model elements in different locations of the system model. The `DSub2BusPort` instantiates the port from the network repository and defines a new attribute `sealed`. The new port type is then instantiated in the configuration repository. Attributes that are read-only are shown gray, new properties red and others black.

connect elements with this network technology, but cannot change its internal attributes (voltage, pins) anymore.

The `extensible at` constraint for the port specifies that model users can define new properties for ports in the specification phase. An example for such an extension is the `sealed` property, which is defined in the `InterfaceTypeCollection` repository. Instances of this `DSub2BusPort`, in the application phase, have to specify a value for this property. Port elements in the application phase cannot be extended with new properties (because its not explicitly allowed via constraint). This way, the modeling environment enables extension of the class facet of domain elements, however, in a configurable way (Requirement R4).

This use-case shows the difference of repositories + phases and levels. With different context models for elements from different domains, repositories do not have a global linear order. Instead, they are free floating and their contained domain elements are in (ordered) phase states depicted by the context model. First-phase elements of different domains do not have to be in the same repository: the first derived element of a camera and port are neither in the same phase nor in the

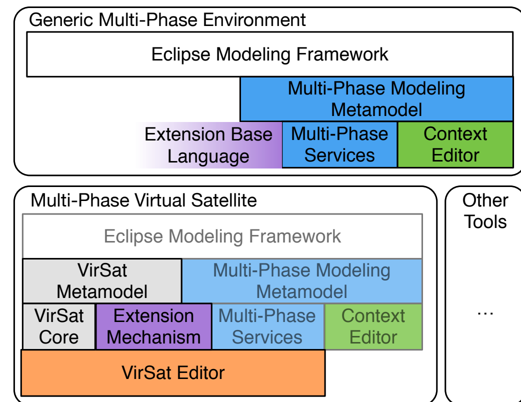


Fig. 15 Structure of the prototype implementation. The modeling environment is domain independent and can also be used in other tools than Virtual Satellite.

same repository. As shown in the figure, elements of different domains are derived following their own dimension. This enables to incorporate several different domains into one system model and thereby fulfills Requirement R3.

5.3 Integration into Engineering Processes

To analyze the impact on tools and the underlying process of modeling, we implemented a prototype for multi-phase modeling. As our approach targets modeling of multiple project phases in interdisciplinary system engineering projects, Virtual Satellite[1] is a suitable tool. It enables modeling space systems and supports engineering for the whole life-cycle of systems [15].

Modeling systems from different abstraction levels with multi-level modeling is not supported in the current version of Virtual Satellite. However, it is using a concept similar to levels to handle configuration problems, such as definition, configuration and assembly of satellites. If elements are reused in different of these 'levels', they are copied. Thus, elements are not customized to the different abstraction levels. Elements in the different 'levels' all have the same attributes.

To change that, we developed a generic environment for context-aware multi-phase modeling and integrated it into Virtual Satellite. Figure 15 shows the structure of our implementation. It defines types for the context model and context-aware constraints (as shown in Figures 6 & 11). To specify context-aware constraints, it contains a textual language definition that implements the specifiers from Table 1.

[1] Virtual Satellite: `https://github.com/virtualsatellite` developed by the Institute for Software Technology at German Aerospace Center: `https://www.dlr.de/sc/en`

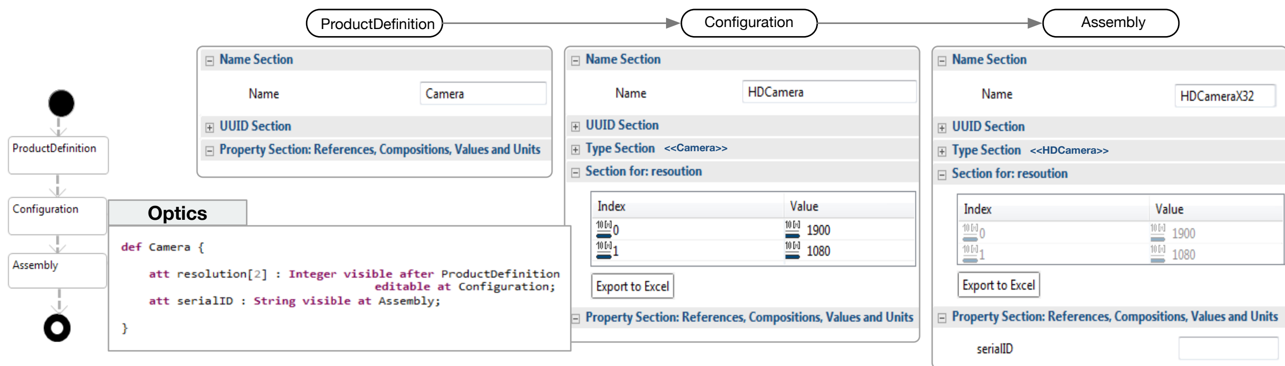


Fig. 16 Diagram editor for the context model, text editor for type-definition and a context-aware UI-editor with a camera element in different modeling phases. As defined by the inline context-aware constraint in the camera definition, the resolution property is visible after the first phase and editable in the configuration phase (not in the assembly phase).

Virtual Satellite provides an extension mechanism that is based on the type-object pattern and uses a textual language to define new types. To support context-aware constraints for these types, we updated Virtual Satellite's extension mechanism to include the language for constraint specification. We, furthermore, updated system model elements to be an implementation of a DerivableElement to enable usage in multiple modeling phases. This way, it is possible to use multi-phase modeling and to define context-aware constraints within Virtual Satellite. While an in-depth description of this implementation is out of scope of this paper, the tool environment uses a realization dimension comparable to the approach described by Igamberdiev et al. to enable deep modeling [23]. Both, the user interface and the model infrastructure, are now extended to be context-aware. The UI only shows elements that are visible according to the context model and context-aware constraints; the underlying model engine checks every change for compliance with the current context.

Figure 16 shows the context-aware Virtual Satellite editor. Most left, the context model is opened in its diagram editor. The text editor screenshot shows the domain-specific language for defining new types in Virtual Satellite. Besides the definition of a camera element, it contains three inline context-aware constraints (inline because they are within the definition of the camera type). The UI editor, then, shows three camera elements in different phases. The most left UI editor screenshot shows the creation of a generic camera element. The camera element in the second derivation step (middle) allows to edit the camera's resolution. The serial id property is only visible in the element of the assembly phase. In a later added integration phase, between configuration and assembly, the editor would not show the serial id and the resolution would be read-only.

The prototype implementation shows that an implementation of context-aware multi-phase modeling has a direct benefit for system engineering tools. It does not only support dynamic changes of the order of phases but it also enables phase-specific modeling. As shown in Figure 16, users of the editor then only see the properties which are semantically correct in the given phase.

Multi-phase modeling provides a unified way how data models can be derived to new development phases. The methodology respects that development phases can require a different level of abstraction. It relieves engineers from re-creating MBSE infrastructure for new development steps, because elements can be derived natively to new modeling phases. As shown in Figure 14, it thereby allows modeling different engineering perspectives: for example, one phase enables specifying the port's 'technology', the second phase enables its application. To achieve backward-compatibility and to avoid superfluous context models, DerivableElements can also be used without context. In such a case, elements are always in a single state defined by an implicit context model. As a result, these elements can be added to all repostitories, but without a life-cycle definition in a context model they cannot be derived. This mechanism enables to use elements that are not relevant in multiple phases without context model. Users have to create a context model only when beneficial for the modeling process. Future work will have to provide quantitative analysis of the modeling effort saved through multi-phase modeling. Independently, the explicit `derived from` relation between elements of different modeling phases improves data continuity. Re-using already modeled aspect is key to improving engineering processes [28]. However, development phases usually require modeling the system from different abstraction levels. Multi-phase modeling is beneficial when classic multi-level modeling is too strict, e.g. when required

phases/levels are not mere classification levels. In engineering, this is can be the case as 'levels' might come from configuration control [37]. Multi-phase modeling can be used with configuration levels by still enabling to trace back, e.g., why a component on a spacecraft was added and how it evolved during the course of the development process.

5.4 Integrity and Limitations

Customization of the context model allows to completely reorder the modeling phases. Such a fundamental change of the modeling process might lead to inconsistencies. However, with the rules for customization of Section 4.2.3, phases cannot be deleted. This removes the risk of dangling references of, e.g., context-aware constraints. New phases can only be added/edited after the currently active phase to ensure that no inconsistency between context and system model appears. Adding phases, as done in the example of, e.g., Figure 13, leads to modeling phases differing between different contexts. The satellite ID in the example of the initial project is modeled in the third phase; in the customized context it is modeled in the fourth phase. From a global modeling perspective, without context model, this might be considered inconsistent. However, with the context model, these changes of the process are formalized and, thus, can be traced. Reordering phases that are referenced by context-aware constraints might lead to 'missing dependencies'. E.g. according to constraints, property X is visible after phase A and can be edited in the following phase B, switching the phases A and B results in X not being visible in B. X could then not be edited in any phase. However, as context-aware constraints reflect the modeling process, such a case also indicates a broken development process. Furthermore, tool-support can validate customization of context models and warn if, e.g., properties are not visible but are configured to be editable.

5.5 Implications of Multi-Phase Modeling

Multi-phase modeling, as described in this paper, was inspired by multi-level modeling but focuses on use-cases with multiple modeling phases. Although the abstraction level of elements decreases along the modeling process, multi-phase modeling is based on more than one abstraction principle. For example, we do not explicitly require the relation of elements between different phases to be only the non-transitive classification one. For our engineering process this does not cause any serious issues, as we do not rely on the classification characteristic in any way. Regardless of the semantics of the relation of elements between different phases, it can at least be used to trace elements throughout the modeling process. However, when using multi-phase modeling, one has to keep in mind that phases differ from levels and, thus, not all multi-level modeling concepts might be directly applicable. We do not preclude, however, that context-aware modeling can directly be used for multi-level modeling. If relations between elements of different phases are enforced to be multi-abstraction relationships, then, phases might actually be seen as levels. Nevertheless, driven by our engineering use-case, we intentionally do not require the relation between phases to be non-transitive. Not having clear and consistent semantics of this relation might make it difficult to discover modeling anomalies and inconsistencies. A solution to this disadvantage can be a parametrization of the `derivedFrom` relation via context model. Specifying that all inter-phase relationships between two specific modeling phases require to be classification relations, allow to run MLM soundness checks on these specific phases. E.g. we could specify that the relation between the `Specification` and `Application` phase in Figure 14 requires all inter-phase relations to be of type `instanceOf`, thus, conforming to MLM rules. Such a parametrization of phase transitions enables to run customized soundness checks on the different phases.

This section applied multi-phase modeling to two examples of the systems engineering domain and presented a tool implementation. Our first example demonstrated that multi-phase models can be used in contexts with different level hierarchies than the base model. It thereby shows that Requirement R.1 is fulfilled. The second example shows that models from different domains can be integrated into one system model and thereby solves Requirement R.3. The attributes of ports are, furthermore, configured to be either editable in the specification or in the application phase and thereby demonstrate the behavior requested by Requirement R.2. The last Requirement R.4 is demonstrated by the `Port` being configured to be extensible in the specification phase only. By modeling the pin-connector engineering example, and showing the integration into the Virtual Satellite engineering tool, we show a direct benefit for systems engineering projects.

6 Related Work

This work uses multi-level modeling concepts and applies them to a development process with multiple phases. In this section, we compare multi-phase modeling with related approaches of multi-level modeling.

6.1 Multi-Level Modeling

The way we utilize multi-level modeling differs from strict implementations, such as by Atkinson and Kühne [9]. Our approach of using multi-level modeling is closer related to Materialization and M-Objects [11][32]. However, the concept of separate metamodeling spaces is strongly related to our approach of evaluating the phase state locally per domain [6]. This local handling of the phase of a domain element also results in the fact that repositories, the structural containers of domain elements, are not ordered in any way. An ordering of instances of a domain element exists only if explicitly specified in a context model. A domain element without a context model (for consistency that is an implicit context model with one phase) can be instantiated in all repositories without further restrictions. Then, it is always in the one state defined by the implicit context model. Such a domain element corresponds to an orderless type in the modeling theory of Almeida et al. [3].

One of the most advanced tools for multi-level modeling is Melanee [4]. Our multi-phase modeling implementation and the *Derivable Element* is based on the concept of a *Clabject*, presented in their work. While it is not in the focus of this work, they provide advanced means to dynamically customize element visualization by also combining textual and graphical domain-specific languages [5]. With a mapping of element properties to modeling phases, as presented in this work, element visualizations can be further customized. A combination of both modeling approaches could allow new customization techniques for phase/level and domain-specific editors. Melanee also contains a constraint language to specify level-spanning constraints, which are aware of the ontological modeling dimension [27]. Concepts, such as potency, however, which are a foundation for these constraints, are based on consecutively numbered level labels. Thus, they do not allow inserting new model levels. Fischer highlights this problem for the domain of systems engineering [14]. Kühne, furthermore, points out challenges of the concept of *potency* for order-aligned model levels and suggests to consider these locally [25]. Using dedicated context models for different domains describing the order of modeling phases corresponds to the idea of different *modeling spaces* and a *total local order alignment*. However, this paper introduces a formalism to explicitly specify and manipulate the expected order of levels/phases of these modeling spaces.

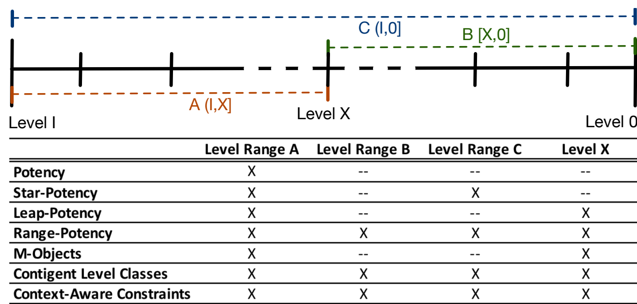
Controlling element extensions, as our approach does it with the extensible constraints, has a related concept in literature: MetaDepth allows controlling weather an element can be extended by two different types of ontological instantiation (*strict* and *extensible*) [12].

Frank presented a concept of domain-specific language hierarchies based on multi-level models [16]. He suggests using languages in different levels of organizations. Abstract languages can be shared between different groups while concrete ones add project-specific details. Such an approach can be supported and implemented by a modeling environment aware of context models. The application domain introduced in Figure 9 implicitly considers such a usage already.

Deep references, as defined by de Lara et al., allow referencing clabjects by their most abstract type [13]. In combination with a potency, indicating the depth of the instance to be referenced, this allows specifying references to concrete instances not yet known. Our approach uses a similar way of referencing other model elements. As shown in Figure 14, references to other model elements can contain a phase specifier (a reference into a context model) to define at which phase a referenced element needs to be. In the example of Figure 14, the `Component` references derived elements of ports by their most abstract type (`Port`), but specifies that these have to be in the application phase.

6.2 Potency

One of the main contributions of this work is to enable phase-specific modeling, thus, explicitly specifying in which phase / level a model element is supposed to be modeled. Foundation for this idea is the initial concept of potency [7]. Our counterpart, context-aware constraints, work on types and properties, as do Single/Multi-potency [36]. However, we do not explicitly differentiate in our terminology. The constraining types `editable` and `visibile` in our apporach, correspond to `mutability` and `durability` in other modeling implementations [19]. However, in contrast to phases, only the non-transitive classification relation is allowed between levels. Thus, with classic multi-level modeling, inserting classification levels between existing ones is impossible. However, the general principle of potency is still related to our context-aware constraints. In the literature, several extensions and customization of the original potency can be found. Figure 17 shows a comparison of these different potency-related concepts. The original potency is based on a number that specifies the depth to which a model element can be instantiated. As shown in the table in Figure 17, this allows specifying that a model element can be instantiated until Level X. Star potency allows to define that an element can be instantiated in an arbitrary number of levels and allows refinement to a regular potency in one of its instances [19]. Thus, the difference between both approaches is that star potency does not require



	Level Range A	Level Range B	Level Range C	Level X
Potency	X	--	--	--
Star-Potency	X	--	X	--
Leap-Potency	X	--	--	X
Range-Potency	X	X	X	X
M-Objects	X	--	--	X
Contigent Level Classes	X	X	X	X
Context-Aware Constraints	X	X	X	X

Fig. 17 Comparison of different extensions of the original potency or concepts that have a similar effect. The table presents if the different potency types can be used to specify if model elements in Level I can appear only in the subset of levels defined by A,B and C.

target committal early on. Star potency corresponds to our phase specifier `always` (Both, star potency and our specifier can be customized later). Leap potency allows to specify that an element can be instantiated exactly X levels from the current element [13]. Intermediate levels are skipped. This potency type corresponds to our phase specifier `at X`. Range potency uses min and max values to specify flexible ranges for instantiation [30]. These ranges allow targeting all the subsets of levels shown in Figure 17. Individual levels can be targeted by using the same min and max value; a * as max value allows instantiation until Level 0. Our level specifier `between X and Y` (Table 1) directly corresponds to range potency. M-Objects use labels instead of level numbers to specify at which level a contingent object has to be instantiated [32]. Contingent level classes allow specifying a 'range of possibly represented levels' and thus also supports all subsets of levels [17]. Note that we further discuss M-Objects, contingent level classes, Join potency and source/target potency in Section 6.4.

Range potency and contingent level classes enable to configure flexible ranges of levels in which model elements can be instantiated. However, using numbers to target levels prevents using this mechanism in environments where additional levels might be added later. This problem is solved with a context model that can be updated when knowledge about further modeling phases arises. The context model provides an ordinal scale for environments where numbers cannot be used because of incomplete knowledge about how many derivation steps are required for an element. Level labeling without a context model (e.g. in M-Objects) does not provide an intuitive ordinal scale: individual labels do not directly show where in the hierarchy they are located. In contrast, constraints with a reference into the context model automatically point to a structure that shows the current ordering of phases.

6.3 Process Modeling

Guerra and de Lara presented a related approach that also explicitly describes the modeling process [21]. Their process targets to improve flexibility when creating multi-level models. The presented architecture supports configurable meta-modeling options to switch between different amount of flexibility. Their process model is used to configure the strictness of the environment in the different phases and to do configurable conformance checks. In contrast, our current approach enforces similar modeling strictness in all phases, but allows to map modeling aspects to the specific phases. However, as mentioned in Section 5.5, it might make sense that the context model could also be used to configure the strictness and conformance checks on the different phases.

6.4 Modeling Hierarchies

There exists multiple related approaches to integrate models with different modeling hierarchies into one model:

A related concept is the Join potency [38]. It also targets to integrate models from multiple domains with different modeling hierarchies. Join potency uses meta-level-pointers to specify which clabjects of the different models have to be joined to integrate them into one multi-level 'megamodel'. In comparison, until know, our approach assumes that no clabjects have to be joined when creating system models. We target to integrate models from different domains, where elements represent different system aspects, and can coexist in the combined system model. This might be too simple for complex systems engineering use-cases where models are overlapping. A future approach could combine join potency with context models to have a strong connection to the modeling process and to join elements where necessary.

Another related concept that uses non-numbered labels for model levels are M-Objects [32]. Similar to context models, their concretization hierarchies describe the order of these levels. While it would be possible to model the telecommand example from Section 5 with M-Objects, their approach does not describe an explicit description of the level ordering. Explicitly modeling the context allows using different context models in projects. This way, a common base model can be used and it is still possible to e.g. rearrange the order of phases in specific projects. Furthermore, the same system aspects can be modeled in different phases of projects by still keeping consistency: the mapping of properties to phases via context model can be used to backtrace what is modeled in the different development phases.

Contingent level classes are elements without an explicit level but with the ability of direct concretization on different levels [17]. Similar to M-Objects, properties can target flexible levels for concretization. In contrast to M-Objects and context models, contingent level classes use numbered levels. To allow instantiation in contexts with different level hierarchies, properties can be assigned to contingent instantiation levels. Contingent instantiation levels enable to refine the level for concretization later and thus enable handling of incomplete knowledge. However, as levels are numbered, it is not possible to use contingent level classes in environments where levels can be added later (if renumbering is not an option).

Dual deep modeling also aims to connect different modeling hierarchies [33]. Like classic multi-level modeling, their approach is based on level numbers. However, different hierarchies can be modeled independently from each other. Properties have a source and a target potency. The source potency is specified relative to the source clabject's hierarchy; the target potency relative to the target clabject's hierarchy. Both have an implicit equivalent in our approach: the source potency corresponds to a context-aware constraint with the property as subject (e.g. 'output editable until configuration' in Figure 14). While the target potency corresponds to the optional phase specifier for referenced types from other domains (See the component inputs / outputs in Figure 14). Furthermore, dual deep modeling allows specialization of deep properties. This enables to customize source and target potency and thereby allows handling incomplete knowledge when creating the property. We do not have a direct representation of that mechanism, however, updating the context model of a referenced element can have a similar effect. Nevertheless, specialization of deep properties, as described by Neumayr et al., goes beyond the handling of properties in our approach. On the other hand, context models and their mechanism of reordering, allow more flexibility within one level hierarchy because it is also possible to e.g. add phases/levels between existing ones.

7 Conclusion and Future Work

In this paper we present an engineering modeling paradigm for development processes with multiple modeling phases. It is based on a context model that describes the order of these modeling phases. Based on this context, it is possible to map aspects to be modeled to the development phases. The process, thereby, allows phase-specific modeling. Thus, it allows to configure, when in a development process, it is possible to model which system aspects. Changes in the development process are handled by updating the context model. For example, new modeling phases can be added by inserting a corresponding phase to the context model. This paper's exemplary evaluation applies the new multi-phase modeling approach to real-life systems engineering uses-cases. It shows that the number of phases can be adjusted to the system's development process and that system properties are only accessible if semantically correct. Furthermore, to support interdisciplinary environments, the modeling approach facilitates integration of multi-phase models from different domains. As the ordering of phases via context model is done separately per domain, system models can integrate multi-phase models from other domains without having to consider a global phase ordering (as it would be in classic multi-level modeling approaches with levels). While this paper presented the conceptual basis for context-aware modeling, future work will present our implementation in depth and also compare it with existing multi-level modeling tools. Quantitative analysis needs to compare the modeling effort with and without multi-phase modeling. It also makes senses to discuss if multi-level modeling in general benefits from context-aware modeling. Future work should, furthermore, evaluate if this modeling approach can be combined with join potency, to enable joining clabjects from different models to be integrated [38]. In addition, specialization of properties, as presented by Neumayr et al., could be integrated to improve reference handling between elements of different domains [33]. This way, integration of models from different hierarchies into one system model can be improved by keeping the strong connection to the modeling process.

References

1. Meta Object Facility (MOF) Specification Version 1.4.1 Formal/05-05-05 (2005)
2. ECSS-M-ST-10C space project management - project planning and implementation (2009)
3. Almeida, J.P.A., Fonseca, C.M., Carvalho, V.A.: A comprehensive formal theory for multi-level conceptual modeling. In: International Conference on Conceptual Modeling, pp. 280–294. Springer (2017)
4. Atkinson, C., Gerbig, R.: Melanie: multi-level modeling and ontology engineering environment. Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards (Lml), 5–6 (2012). DOI 10.1145/2448076.2448083
5. Atkinson, C., Gerbig, R.: Harmonizing Textual and Graphical Visualizations of Domain Specific Models. In: Proceedings of the Second Workshop on Graphical Modeling Language Development - GMLD '13. ACM, Montpellier (2013). DOI 10.1145/2489820.2489821
6. Atkinson, C., Kühne, T.: Processes and products in a multi-level metamodeling architecture. International

Journal of Software Engineering and Knowledge Engineering **11**(06), 761–783 (2001)
7. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: ≪UML≫ 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools, vol. 2185, pp. 134–148. Springer, Berlin, Heidelberg (2001). DOI 10.1007/3-540-45441-1
8. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation **12**(4), 290–321 (2002). DOI 10.1145/643120.643123
9. Atkinson, C., Kühne, T.: Concepts for comparing modeling tool architectures. In: International Conference on Model Driven Engineering Languages and Systems, pp. 398–413. Springer (2005)
10. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and Systems Modeling **7**(3), 345–359 (2008). DOI 10.1007/s10270-007-0061-0
11. Dahchour, M., Pirotte, A., Zimányi, E.: Materialization and its metaclass implementation. IEEE Transactions on Knowledge and Data Engineering **14**(5), 1078–1094 (2002)
12. De Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: International conference on modelling techniques and tools for computer performance evaluation, pp. 1–20. Springer (2010)
13. De Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending deep meta-modelling for practical model-driven engineering. The Computer Journal **57**(1), 36–58 (2014)
14. Fischer, P.M.: Potential of Multi Level Modelling in Model Based Systems Engineering. In: Dagstuhl Seminar 17492 (2017)
15. Fischer, P.M., Lüdtke, D., Lange, C., Roshani, F.C.., Dannemann, F., Gerndt, A.: Implementing model-based system engineering for the whole lifecycle of a spacecraft. CEAS Space Journal **9**(3), 351–365 (2017). DOI 10.1007/s12567-017-0166-4
16. Frank, U.: Multilevel Modeling. Business & Information Systems Engineering **6**(6), 319–337 (2014). DOI 10.1007/s12599-014-0350-4. URL http://link.springer.com/10.1007/s12599-014-0350-4
17. Frank, U., Töpel, D.: Contingent level classes: motivation, conceptualization, modeling guidelines, and implications for model management. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–10 (2020)
18. Galvão, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC, pp. 313–324 (2007). DOI 10.1109/EDOC.2007.4384003
19. Gerbig, R., Atkinson, C., De Lara, J., Guerra, E.: A feature-based comparison of melanee and metaDepth. CEUR Workshop Proceedings **1722**, 25–34 (2016)
20. Gross, D., Yu, E.: Evolving system architecture to meet changing business goals: an agent and goal-oriented approach. In: Proceedings Fifth IEEE International Symposium on Requirements Engineering, pp. 316–317 (2001). DOI 10.1109/ISRE.2001.948602
21. Guerra, E., de Lara, J.: On the quest for flexible modelling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 23–33 (2018)
22. Hinkel, G.: Using structural decomposition and refinements for deep modeling of software architectures. Software & Systems Modeling **18**(5), 2787–2819 (2019)
23. Igamberdiev, M., Grossmann, G., Selway, M., Stumptner, M.: An integrated multi-level modeling approach for industrial-scale data interoperability. Software & Systems Modeling **17**(1), 269–294 (2018)
24. Johnson, R., Woolf, B.: Pattern Languages of Program Design 3, chap. Type Object, pp. 47–65. Addison-Wesley Longman Publishing Co., Inc. (1997)
25. Kühne, T.: A story of levels. In: MULTI 2018. Copenhagen (2018). URL https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/10/multi_paper6.pdf
26. Kühne, T.: Exploring potency. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 2–12. ACM (2018)
27. Lange, A., Atkinson, C.: Multi-level modeling with MELANEE. In: Proceedings of the MULTI 2018 (2018)
28. Lange, C., Grundmann, J.T., Kretzenbacher, M., Fischer, P.M.: Systematic reuse and platforming: Application examples for enhancing reuse with model-based systems engineering methods in space systems development. Concurrent Engineering **26**(1), 77–92 (2018)
29. Lyardet, F.D.: The Dynamic Template Pattern. In: Proceedings of the Conference on Pattern Languages of Design (1997)
30. Macías, F., Rutle, A., Stolz, V., Rodriguez-Echeverria, R., Wolter, U.: An approach to flexible multilevel modelling. Enterprise Modelling and Information Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 13, Nr. 10 (2018)
31. Neumayr, B., Grün, K., Schrefl, M.: Multi-Level Domain Modeling with M-Objects and M-Relationships. Conference on Conceptual Modelling **96**(Apccm) (2009)
32. Neumayr, B., Schrefl, M., Thalheim, B.: Modeling Techniques for Multi-Level Abstraction. In: R. Kaschek, L. Delcambre (eds.) The Evolution of Conceptual Modeling, vol. 6520, pp. 68–92. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-17505-3_4
33. Neumayr, B., Schuetz, C.G., Jeusfeld, M.A., Schrefl, M.: Dual deep modeling: multi-level modeling with dual potencies and its formalization in f-logic. Software & Systems Modeling **17**(1), 233–268 (2018)
34. Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. Journal of Systems and Software **111**, 272–280 (2016). DOI 10.1016/j.jss.2015.08.047
35. Rao, B.H., Padmaja, K., Gurulingam, P.: A Brief View of Model based Systems Engineering Methodologies. International Journal of Engineering Trends and Technolog **4**(8), 3266–3271 (2013)
36. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. Formal Aspects of Computing **26**(6), 1115–1152 (2014)
37. Svensson, D., Malmqvist, J.: Strategies for product structure management in manufacturing firms. In: International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, vol. 35111, pp. 377–386. American Society of Mechanical Engineers (2000)
38. Theisz, Z., Bácsi, S., Mezei, G., Somogyi, F.A., Palatinszky, D.: Join potency: a way of combining separate multi-level models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–5 (2020)