TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur für Softwaretechnologie

Master Thesis

User-Driven Constraint Modelling for Entity Models at Runtime

Anton Skripin

Born on: 10.03.1998 in Novouralsk, Russia

04.11.2023

Supervisor

Dr.-Ing. Sebastian Götz

Supervising professor

Prof. Dr. rer. nat. habil. Uwe Aßmann

Statement of authorship

I hereby certify that I have authored this document entitled *User-Driven Constraint Modelling for Entity Models at Runtime* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 04.11.2023

Anton Skripin

Abstract

Here should go my abstract

Contents

1 Introduction

This chapter introduces the problem space and incentives for this work to appear. Section 1.1 reflects the key motivation by providing a chronological overview of a modeling evolution, starting with the basic notion of a model and finishing with the modern challenges focusing on models@run.time. Section 1.2 states current gaps in the research community by introducing a concrete example. Section 1.3 familiarizes a reader with the main contributions and research questions, the answers to which should be found in the scope of this work. Finally, section 1.4 gives offers a summary for every following chapter.

1.1 Motivation

The formalized notion of a model emerged from the model theory. According to the model theory and omitting all complexity, a world consists of objects. In turn, objects are endowed with properties. A model represents an original whose properties are described as mapping to the image in a model. [CK90] It is worth noting that not every model is an abstraction of a real object but can be an abstraction of another model. An intuitive example of such a sequenced model is a map application, the domain model of which cannot be a direct model of our planet. Furthermore, as stated by [Hel+16], depending on the purpose of a model it serves, it can be either descriptive or prescriptive. The former means abstracting a real object. Hence, an origin stems from an actual entity. The letter ones comprise the specification of a real entity to be constructed. Deviation from a prescriptive model specification indicates an error. Thus, in such models, an origin originates from the specification of the created entity.

Modern software systems are non-trivial and consist of numerous artifacts. Thus, requirements are collected from involved stakeholders and reflected in design and system architecture. The source code then manifests all the stages before. Finally, the documentation for a system is created either manually or automatically to maintain the origin of software knowledge among involved parties. One vital concept to grasp, however, is that nowadays, a final software product is much more than just a program code. All system artifacts are necessary to produce and support a system during its software lifecycle. If the final software lacks at least one of the elements mentioned earlier, it cannot be regarded as software but just as some purpose-specific script.

Nevertheless, what overarching role do models play during the development of software? First, regardless of the design paradigm for the development of software systems, a model is a link between a client and a developer that serves as an intermediate component every involved party can understand. Secondly, models cooperatively with documentation help keep a system's essence during its evolution.

One of the most natural ways to present and manage something complex is to depict it through modeling. Models can have different purposes. They can be applied to depict a desired structure or behavior of a system before development. Besides, models are perfect for deriving system attributes during or after development to grasp its functioning better. Thus, an object-oriented data model [Day90] must bridge a semantic gap between the real world and relational tables. On the other hand, relational models [Cod07] are highly used in database management to help experts characterize and handle data stored in a database. Being one of the most common and adopted modeling languages, UML (Unified Modeling Language) [Rum05] finds itself at the heart of Model Driven Architecture (MDA). [Sol+00]

Driven by models, MDA aims to facilitate the design and development of modern systems via weaving, traceability, transformation, and automation techniques [Sol+00]. The models produced within MDA showcase the development cycle of a system employing structural or behavioral models. As mentioned by [FR07], these models form a layer of abstraction above the code level. However, in recent years the demand is rising to casually connect a model with its running system. A potential area of application are safety-critical systems that must guarantee high availability and responsibility even in the presence of errors. Coupling a model with a system state would allow the system to adapt its behavior without downtime. Runtime models also highly impact developers and end-users. Consequently, software engineers would benefit from this concept by being able to fix design flaws or introduce new components at runtime. Furthermore, models@run.time would give end-users the advantage of changing a service bahavior dynamically. [BBF09]

Despite a rich scope of application and increasing potential for use in the future, models are often not explicit enough to provide full expressiveness and manifest all functional and non-functional requirements. One of the ways to circumvent such a limitation is to use a constraint modeling language to enrich a model with functional and non-functional requirements. Up until nowadays, the research community has been looking for various means to provide full expressiveness while modeling. However, the current state of research does not offer a bridge between the power of constraint languages and dynamic and variant-aware models at runtime. Therefore, certain questions have arisen and, up until now, unanswered. Does it even make sense to enrich runtime models with constraining capabilities? If the answer is yes, how must a software system look at the end where constraints enrich the concept of models@run.time?

1.2 Problem Statement

As a model-centric technique, MDE has proven to be effective in developing high-quality software with improved cost-effectiveness. However, the MDE approach does not support software evolution. Indeed, to include a new adjusted requirement to software, one would need to go through the whole model transformation process and redeploy the system. Unfortunately, such a practice contradicts the high-availability requirements of many software solutions. However, if one casually links a model and a running software product, it would be possible to vanish the barrier between development and runtime artifacts. A promising solution to such a matter is to replace static with runtime models, which perceive models as running software artifacts. [FS17; May+17]

Two sequence diagrams below showcase a simplified snippet from the lifecycle of software. Figure 1.1 presents a well-known MDE approach, figure 1.2 demonstrates a more advanced workflow with the support of runtime models.

Both sequences start with the demand of consumers for new software. Consumer, in this case, represents either a client or a final user to make the diagrams more compact. Then the team using model-driven techniques, creates a final platform-specific model. Creating a model involves gathering requirements, analyzing, designing, and implementing it. For simplicity reasons, this chain of processes is encapsulated into one arrow. With the model, an environment

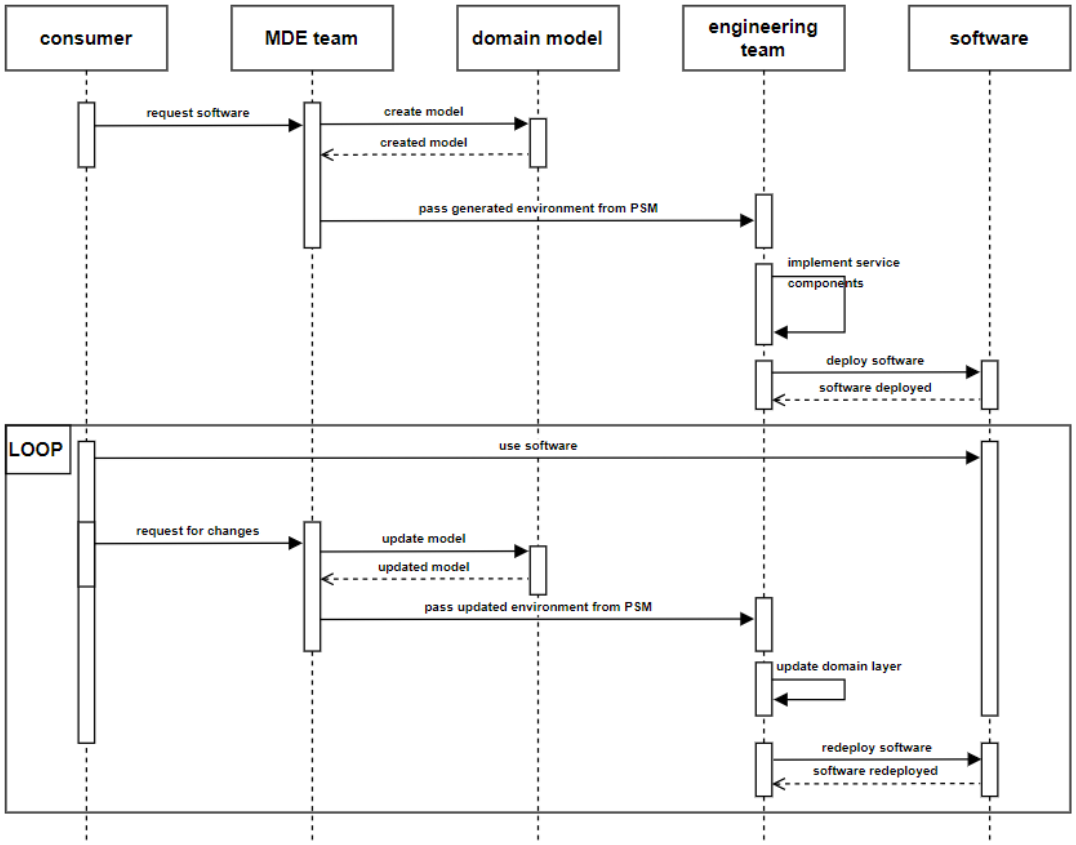

Figure 1.1: Fragment from software development lifecycle with MDE

with an auto-generated domain layer is created and passed to the team of software engineers. They, in their turn, finish the software by implementing missing services and infrastructure. Finally, the software gets deployed and stays available for users. The second technique, however, requires an additional step, i.e., creating a bidirectional link between a model and a running system to reflect and update changes in both directions. Figure 1.2 highlights this step in green color.

Those figures also share the same recurring block that represents software maintenance. The critical difference between these two figures is that whenever a new requirement appears, the software with models at runtime can adjust its behavior dynamically and support continuous, uninterrupted system availability. In contrast, the traditional approach requires software redeployment that could poorly affect end-users. Moreover, as seen in Figure 1.2, in comparison to Figure 1.1, small changes that demand automatic model reconfiguration lessen the burden on software engineers, thus allowing them to focus on more complicated infrastructure in software. We are aware that not only runtime models trigger synchronization in software, but also the software might change, and those changes must be reflected in its backing model. However, to be consistent between two sequence diagrams, only a one-way change from the models@runtime to the software is shown. Furthermore, for the sake of unsophistication, it is also demonstrated that only the MDE team is in power to update the runtime model. However, this action could theoretically be performed by end-users, domain experts, or even the system if it is self-adaptive.

Despite showing significant advantages against traditional model-driven software development, models@runtime should also address the problem of functional and non-functional model constraints during software evolution. As such, figure 1.2 reflects the affected sequence actions in red color.

To elaborate the problem space further, figure 1.3 shows a simple project management sys-

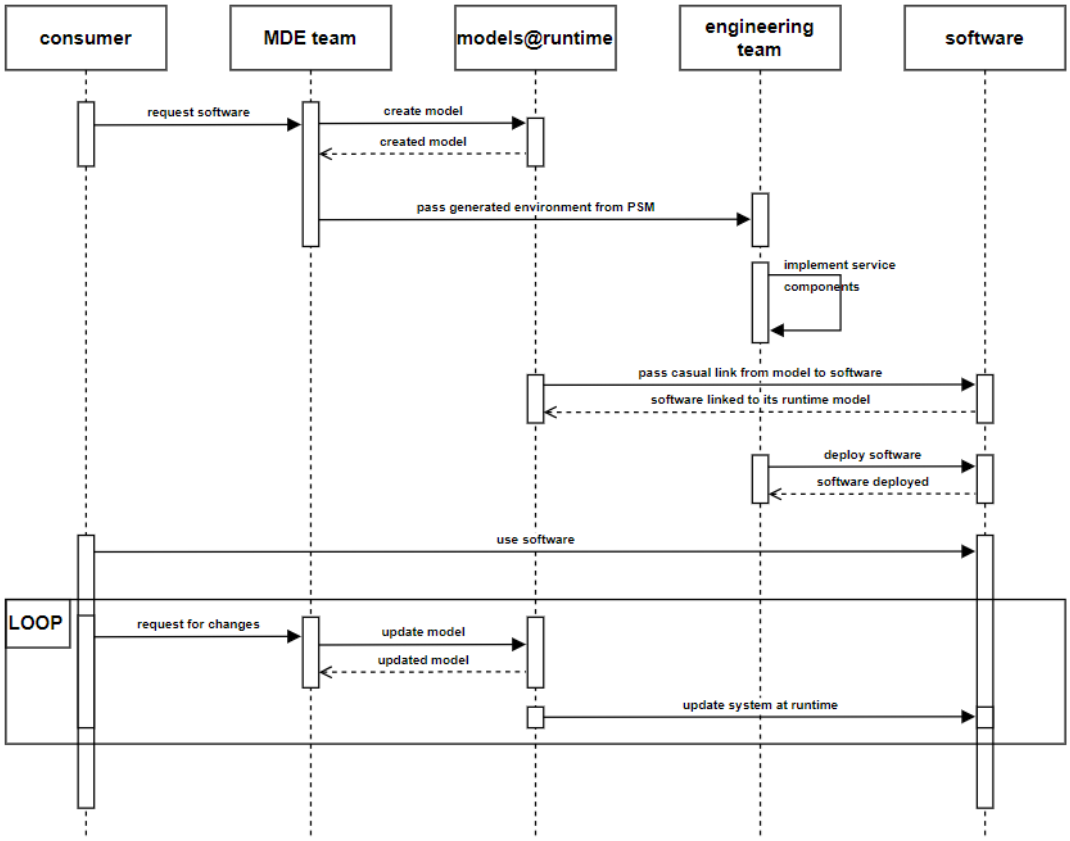

Figure 1.2: Fragment from software development lifecycle with Models@runtime

tem model that reflects a running system. Even though relationships among elements already express some restrictions, they cannot fully describe the domain. For instance, based on the model, it is possible right now for a software engineer to manage a closed project which does not make sense. Consequently, the constraint written in OCL below eliminates this flaw.

```
Context: SoftwareEngineer
inv: self.manages -> forAll(project | project.isClosed = false)
```

The first issue that arises here is constraint-declarative related. As mentioned above, various groups with distinct levels of expertise might need to adapt a model at runtime. It is clear that people with no programming expertise might struggle to constrain the domain in full using declarative tools adhering to some syntax. Similar to how model-driven-development tackles to ease the process of software development, it would make sense to raise the abstraction level for modeling constraints to avoid stakeholders dealing with raw syntax specification.

The second problem stems from the dynamic nature of runtime models. If at some moment in time T, there is a new requirement to introduce a project state, as shown in figure 1.4, then the aforementioned constraint becomes invalid since it refers to the element attribute that does not exist anymore in a new variant. All this might bring inconsistency to the whole system because instantiation mechanisms no longer abide by an initial model specification.

1.3 Objective

This thesis attempts to fill the current research gap in the area of runtime models. Therefore, this thesis aims to research and evaluate the viability and possibility of using constraints

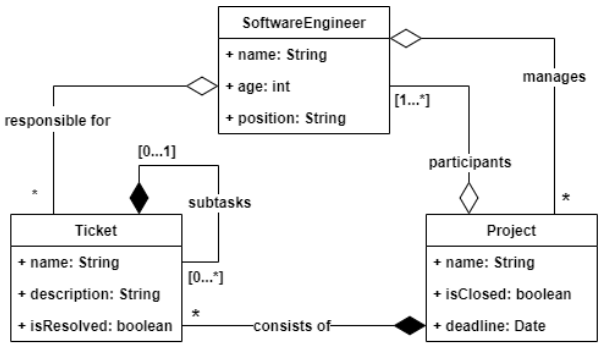

Figure 1.3: Trivial management system class diagramm

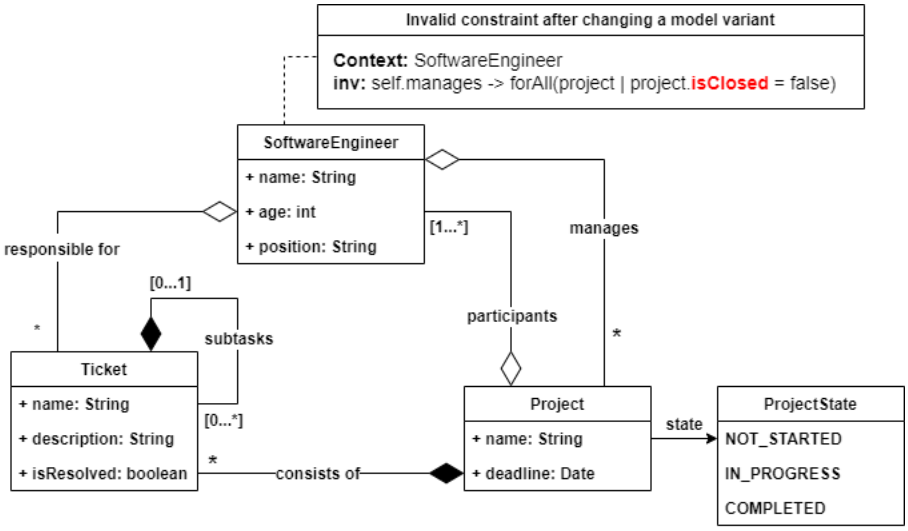

Figure 1.4: Updated model variant results in invalid constraint

for models at runtime by end-users. Subsequently, this work summarizes state-of-the-art regarding various available languages and modeling techniques that should be a prerequisite for overcoming current limitations mentioned in the previous section. Finally, this thesis introduces a framework that allows end-user to impose constraints in an intuitive manner on a runtime variant-aware model.

Furthermore, the main contributions of this master's thesis to the research community are listed below:

1. Introduced the concept of end-user-driven constraint modeling for models at runtime as a specification for restricting variant-aware models by users with no preliminary background in software engineering.

2. Provided a solution to defining such constraints by end-users and keeping the constraints valid in the presence of concurrently existing entities with different variants.

3. Implemented a framework primarily focusing on the novel concept of end-user-driven constraints at runtime. While many frameworks allow users to define constraints in a quite intuitive manner, none of them, to the best of our knowledge, tackles the challenges of shifting the definition and validation of constraints from the design to the runtime state of a system.

Finally, the answers to the following primary research questions are provided as a result of this work:

1.
2.
3.

2 Foundation

This chapter presents the necessary concepts and techniques required for a reader to understand the context of this work.

2.1 Models and metamodels

The use of modeling spans various domains. Many representatives of different areas of occupations, like sociologists, physicists, chemics, mathematicians, and others, create models in order to abstract some concepts and, by doing so, facilitate the understanding of them. Without any doubt the increasing software complexity and the need to exchange information between domain experts and developers, models also play a crucial role in information systems. However, what is a model if we talk about it in common sense? According to [Bro04], a model is an abstraction that allows experts to reason about a system by focusing on relevant and neglecting secondary properties.

2.1.1 Categorization of models

The research community tends to divide models into different purposes of utilization according to their purpose. Although the objective domain of models to be applied varies depending on the field, three main groups of models can be highlighted. The first group helps to grasp some domain knowledge and documentation of software artifacts. The following aids in presenting a system under study and depicting its characterization. The last one allows modeling architecture, design, and platform-specific software concepts in a domain [KW07]. For instance, the combination of all three models is widely used in model-driven development to represent different views and stages of system implementation.

As stated in [BCW17], modeling is impossible without having proper tools. Therefore, two classes of modeling language are distinguished by the scope of application to allow the creation of models.

1. General-purpose-modeling languages (GPLs). This class of modeling languages can describe any domain or specialization. Some of the representatives of this kind of language are UML, Java, SQL, etc.

2. Domain-specific languages (DSLs). They usually feature a more limited syntax in comparison to GPLs. Nevertheless, since any domain language serves a particular need and domain, its power is in its ability to be concise while still expressive.

Nevertheless, the broad distinction between GPLs and DSLs is rather subtle. Depending on the point of view, any GPL can also be viewed as a DSL. For instance, if one determines to apply UML to the domain of all software systems, then it could be regarded as a DSL. However, it also works the same way in the opposite direction. If a domain-oriented DSL starts repeating many flavors of GPM languages, then its benefits are reduced to a minimum.

On the other hand, a functional distinction between DSLs and GPLs is noticeable. A generalized language imposes some limitations on an engineer to describe a domain. The reason is that they are limited to the syntax and metamodel of a language. On the other hand, domain languages significantly foster the software development curve by focusing first on domain-specific concepts. Unfortunately, creating a custom DSL is not always an option due to the following considerations [Sel07]:

1. refining a language to a particular domain requires a lot of resources
2. lack of adequate tool support required for developing software, while comprehensive languages would support a compiler, debugger, and transformation tools for integration
3. lack of support and integration of third-party libraries, unlike general-purpose languages

For this reason, UML provides generic extension mechanisms to be tuned for a domain. Despite being able to specify new language families via profiles, stereotypes, and tagged values, UML remains vague and is not flexible enough to provide full expressiveness for a domain [BD07].

To summarize, modeling plays a crucial role in modern software creation. However, there is no rule of thumb on whether a company should engage in creating its custom DSL or just use GPL. As mentioned earlier, both types of languages possess advantages and disadvantages, and a tool choice might depend on various factors.

2.1.2 Essense of metamodeling in software development

Complex software systems nowadays tend to be characterized by multiple models residing on different abstraction levels. An overarching metamodel enables support for model visualization, semantic definition, and transformation tools [Bro04]. Creating a metamodel means defining a set of common elements used by instances of such meta-elements. In other words, a metamodel is a model of a model [KW07]. As such, UML's whole notation and description are captured in its metamodel, which defines the core elements such as class, association, multiplicity, and connections. The aspects of a UML metalevel enable end-users to describe a system via a set of components and relationships among them.

Objects that live and interact with each other in a runtime environment are described by models. In its turn, to endow models with behavioral semantics and be understandable by machines, models are characterized by formal languages. Consequently, formal languages are denoted by context-free and context-sensitive graph schema, namely by meta-meta models [JB06]. It can be observed that software adheres to some ordered layered infrastructure. Such infrastructure is also known as meta-hierarchy.

Figure 2.1 illustrates a classical four-level meta-hierarchy that comprises the following layers [ZX16]:

- M3 - top layer that resides a meta-meta model or a graph schema of a language. Meta-meta model contains concepts and rules for its instances on a lower level. Usually, this level is self-descriptive. Self-descriptiveness or being lifted enables a language on this level to define itself. That is why there are no M4 nor M5, etc. levels since they would be redundant and would not bring any extra semantics to the concept

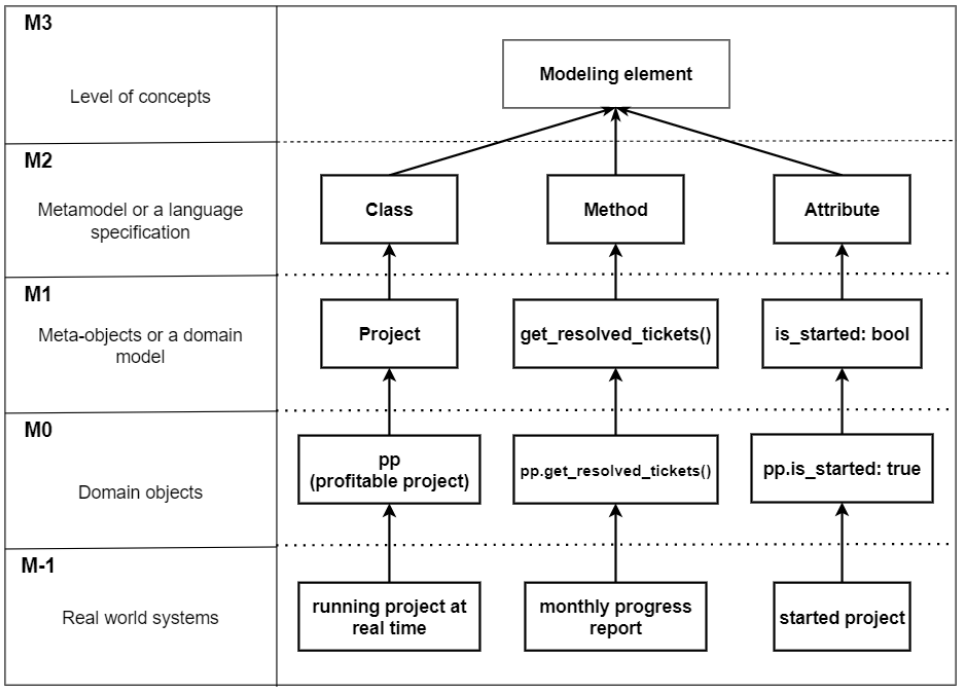

Figure 2.1: Classical meta-hierarchy with digital twins

- M2 level defines types and language specifications. It is an intermediate layer conforming to the M3 layer and serves as a basic infrastructure for the concrete models below. Tools, materials, DSLs, and GPLs are implemented here
- M1 has software classes or application concepts. This level can only have elements and attributes defined on the M2 level. Here could reside a concrete entity class diagram with relations
- M0 level contains real objects that are instances of the M1 level. Similarly to all top levels except the last one, elements of this level are restricted by defined classes on the M2 level
- M-1 level represents a real world. Although it is not part of the classical meta hierarchy and, therefore, not shown in the figure, this level is essential to catch the relationship between natural objects and their digital twins instantiated on the M0 level. A digital twin aims to improve the functioning of software by collecting real-time data from multiple data sources at runtime [Bor+20]

In conclusion, it is essential to grasp the relationship between a model and a metamodel. Both models express distinct contexts that do not overlap, but one can be nested into another. According to the previous statement, Figure 2.2 presents a connection between a model and a metamodel. As such, a model on the M1 level contains a single entity named "Film" that comprises a "Title" attribute. In order to make the definition of such an entity possible, there must exist one metamodel that defines the semantics of the "Film" entity. Therefore, on level M2, the defined concepts "Attribute" and "Class" are created that store the semantics for the below layer. [BG01] The interrelation between the two levels is commonly called the "instance-of" relation [AK03]. The significance of the "instance-of" relation is that any model entity always finds its definition in its metamodel.

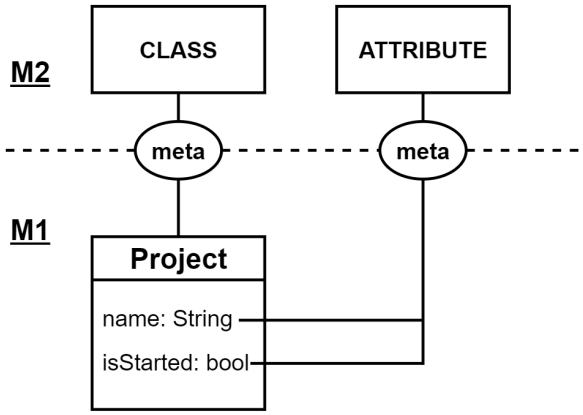

Figure 2.2: Metamodel to model relation

2.2 Model-driven Engineering

Creating modern software involves several continuous and linked to each other stages. Those stages represent the software lifecycle, from gathering requirements to maintaining software and its final decommissioning. If every step of developing software does not map to the previous and the following stages, the launch or further development of such a program is a big question mark. For instance, what is the chance that a discovered bug will be fixed without causing other flaws in a system if the code base does not reflect architecture, domain, and requirements? It is a rhetorical question, the answer to which depends on the size of the software product under maintenance. Nevertheless, as claimed by [HT06], a feasible approach to avoid such issues is to have an overarching meta-concept that allows us to connect all stages of software development and transform different views of software via round-trip engineering.

A promising solution to overcome new challenges of developing industrial software systems is Model Driven Architecture (MDE). The continuous increase in software complexity, the ongoing demand for higher product quality, and the requirement to reduce time to market are the main milestones that identified the need for automation tools while implementing complex software systems [HT06]. An MDE infrastructure should support utilities for model interchange and user mappings from models to artifacts. Metamodeling is highly advantageous to provide such support because it permits model transformation between different abstraction levels derivation of new models from metamodels [AK03].

As claimed in [Béz05], MDE is a technique that encourages the use of modeling languages to provide multiple abstraction levels of a system and facilitate the development process. A software product developed through MDE principles might look as follows and is depicted in Figure 2.3. A software product comprises software platforms, a set of executable models for business process automation, non-generated artifacts written by engineers, and generated artifacts created by transformation tools. Every component is described below in more detail.

First of all, software platform involves reusable components used for creating systems. For example, it could be libraries, middleware components, frameworks, or infrastructure to interact with third-party services. Therefore, it is common to have software that depends on other platforms to encourage the reuse of artifacts. Secondly, generated and non-generated artifacts are also components of a software application. The implication of those artifacts may vary from development time to runtime. The next element of MDE involves transformation techniques. There are two types of transformation - a model-to-text transformation for generating executable artifacts and a model-to-model transformation for consequently deriving models closer to a domain. Finally, as it was mentioned above, models play a central part in the creation of software in MDE. Models can be manually created by domain experts, design-

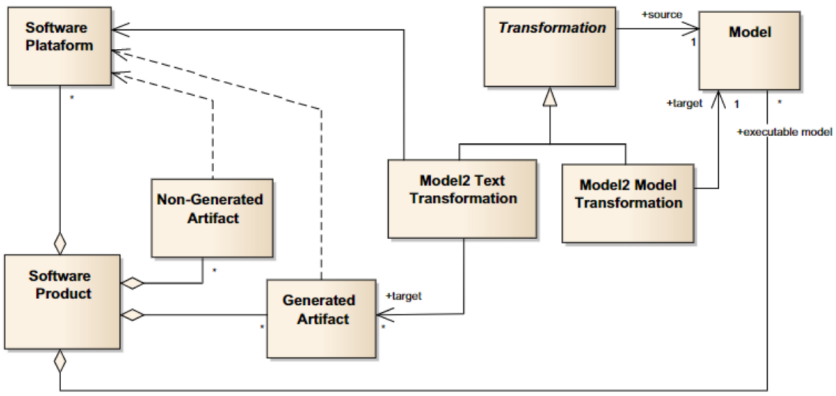

Figure 2.3: Software product composition with MDE, taken from [Da 15]

ers, etc., or generated automatically via transformation techniques. Therefore, a model is a fundamental block to derive generated and not generated artifacts [Da 15].

Model Driven Architecture

One of the pioneers in applying broad MDE concepts was the Object Management Group (OMG). The OMG group introduced the MDA to support the process of creating software ranging from system requirements to software implementation. MDA classifies multiple levels of system abstraction. Each abstraction represents a model with an abstract view of a system. Figure 2.4 shows the levels of artifact generation with the MDA technique and can be classified by the following steps [HT06]:

1. Computation-Independent Model (CIM) - the highest level of abstraction where models present only business rules and domains without assumptions about technological space

2. Platform-Independent Model (PIM) - a level that captures the internal structure of a system and its design with no focus on software infrastructure. Moreover, one PIM can usually derive an arbitrary number of models on lower levels, namely PSMs

3. Platform-Specific Model (PSM) - a level that enriches the PIM model with the functionality relevant to a specific platform. The further usage of PSM should result in concrete source code generation or other executable artifacts

Transformation tools like QVT or ATL do the transition from more to less abstraction in MDA. Manual transformation is also possible but should be disregarded in favor of automation [Da 15].

MDA initiated the transition from code-based to model-based software development. It induced the release of many languages used to specify a software domain. The variety of domain-specific languages encouraged the creation of a unified framework that all meta-languages could conform to and, thus, make them interchangeable. That was one of the reasons for designing the Meta Object Facility language (MOF) - a meta-meta language for all metamodels [Béz04].

However, even though MDA is regarded as a paradigm switch from the long-established opinion that objects play a crucial role in software development [Béz04], it still has some weaknesses and limitations. For example, as noted by [OA15], model software products are requirement-driven. Unfortunately, OMG has no precise formalization about the transformation of gathered requirements from a computation-independent to a platform-independent

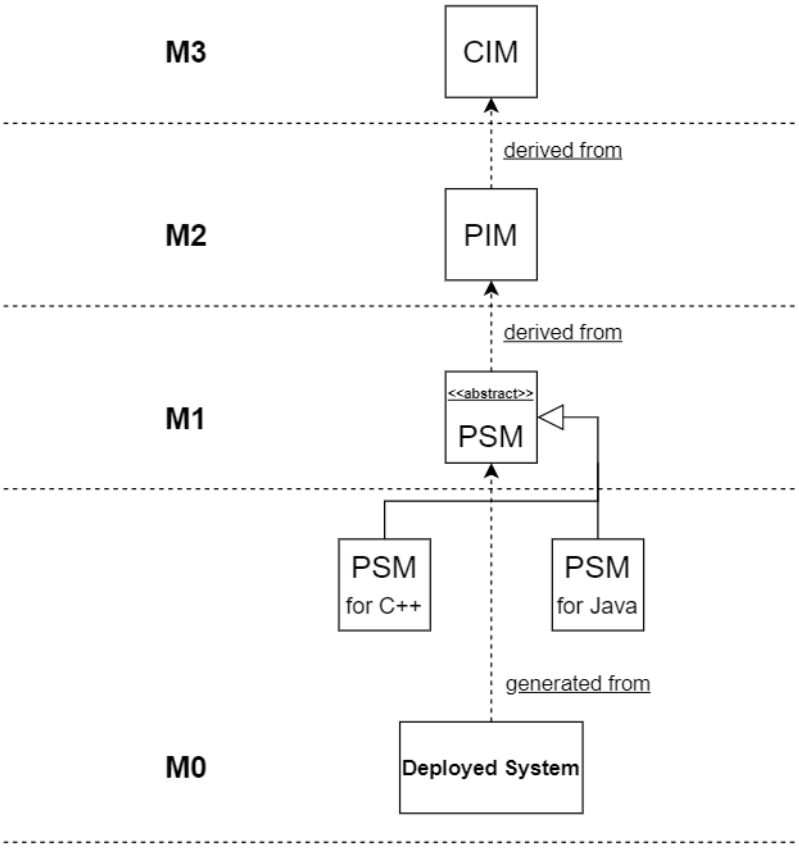

Figure 2.4: Artifact generation with MDA

view. Another author [HT06] argues that the MDA imposes an abundance of multiple distinct abstraction views on a system that requires much effort to keep various models synchronized. Moreover, the problem persists regarding round-trip engineering when the semantics from changed low-level models, such as code or executable models, is complex to map to high-level models. Finally, the author reasons whether MDA actually reduces complexity or only postpones it to some later point since a large amount of produced artifacts and relations among them potentially increases the complexity of tools to be dealt with during software development.

2.3 Multilevel models

Many MDE modeling tools based on MOF as their overarching meta-metamodel need to improve the ability to better express domain needs. The demand for using modeling tools to comply with dynamically changing requirements and creating small but expressive domain languages is rising. However, the classical metamodel hierarchy in the core of modeling artifacts often presents obstacles via their limited support for the extension. As summarized by [AGF13] and elaborated below, the standard extension mechanisms might solve the problem, but all of them tend to increase the overall complexity of a language.

1. Using existing built-in mechanisms for an extension. This approach is quite common among modeling tools since many do not provide direct means of extending a model but instead offer embedded built-in mechanisms. An example of such an approach is UML with stereotypes. Stereotypes allow enriching a target model closer to a required domain. However, it implies limitations on compatibility if every modeling language de-

fines its set of extensions without following some universal convention specified by meta-metamodel.

2. Direct metamodel extension. This approach involves the direct extension of the metamodel of a language. However, the changes cannot be applied at runtime by modeling language after modifying a metamodel. Instead, these tools should be recompiled and redeployed on every change. It results in an additional requirement to migrate models to guarantee synchronization among multiple tools.

3. Model annotation approach. This mechanism enables the separation of an extended language from a language used for model enrichment. The newly introduced enriched models are linked to a target model without direct modifications. Two limitations are apparent in this case. First of all, it requires the support and maintenance of two different models. Secondly, some model elements might have to be duplicated, complicating the final composed metamodel maintenance.

The strict separation of class and object elements is another issue of a classical metamodeling hierarchy. That leads to the problem of "shallow instantiation" [AK01] when an instance of a class can only take the semantics of the direct class used for instantiation. In other words, the two-level instantiation strategy fails to pass information among more than one level in a meta-hierarchy. The works related to multilevel modeling [AGK09] serve promising approaches to overcome such issues. However, since this research field's scope is too broad, this section will only summarize concepts relevant to constraining deep entity models and their implications.

Dual classification of modeling elements

Depending on the point of view, the same element can have traits of both a class and an object. Providing an analogy with a vehicle manufacturing plant, a concrete object Volkswagen Polo is an instance of family classes under the brand Volkswagen. Volkswagen is an instance of all possible cars. Finally, a car is an instance of all likely vehicles produced by our nonexistent fictional factory. This class hierarchy is present in Figure 2.5. One can observe that the elements are instances of some abstract syntax (linguistic type) in which they are defined and are instances of domain-relevant concepts (ontological type). The presence of two instantiation types is known as dual classification. This concept was mentioned by [AGK09], where the author claims that most modeling tools focus only on linguistic instantiation and provide little utilities to express ontological hierarchies across multiple meta-levels. The Orthogonal Classification Architecture (OCA) aims at providing equal support for both types of instantiation by defining two orthogonal dimensions with two different views on a model. Thus, the language dimension views a model as a part of a classical metamodel hierarchy. In contrast, the ontological dimension contains the hierarchy of a domain.

Clabjects and potencies

Another property depicted in Figure 2.5 is the fact that instantiated elements can play the role of both objects and classes. Therefore, such elements have classifier types and instance-of types. An exception is instances residing on the ground M0 level that lack the classifier type to prevent further multilevel instantiation. Moreover, a non-negative integer number placed over attributes and associations shows the number of levels over which a particular element can be instantiated. To pinpoint the duality of model elements and their deep classification, Atkinson [AK01] coined the terms clabjects and potencies.

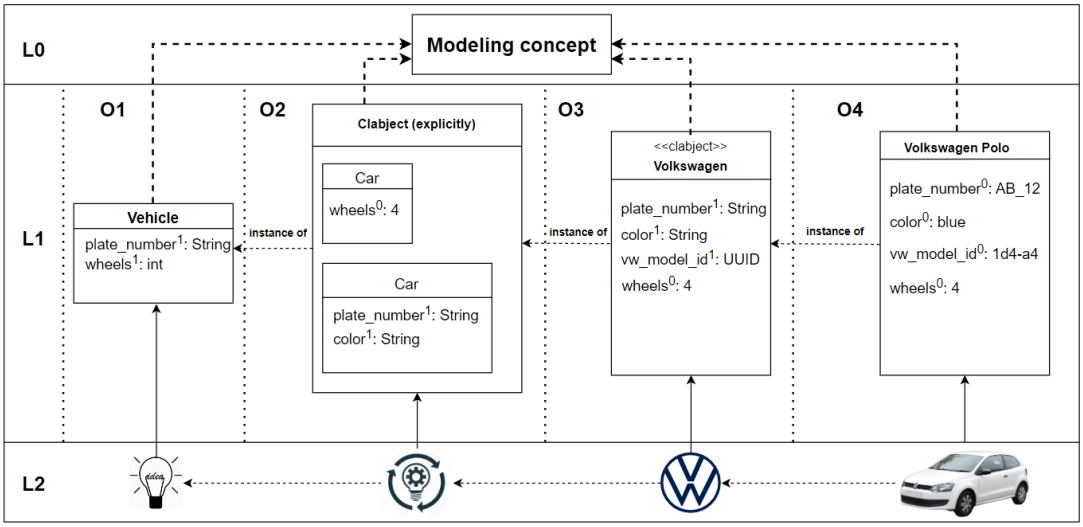

Figure 2.5: Deep model of a vehicle manufacturing plant

Deep constraint language

Current constraint languages should be refined since most modeling tools focus only on the linguistic type of a model. Instead, to constrain a domain where model elements embody attributes and associations from several parent classifiers, a constraint language should be aware of the ontological semantics of clabjects and deep instantiation mechanisms [AGK09]. An example of a possible constraint to our exemplary domain that considers a multilevel notion of an instantiated model element is shown below. Nevertheless, a more detailed classification of constraint languages and their importance in entity model hierarchy is elaborated in one of the following chapters.

2.4 Models at runtime

The emergence of new types of software that operate in highly dynamic environments imposes more requirements on those systems to be adaptable and reconfigurable. Moreover, due to the high dynamism and often unpredictable workflow, these systems are susceptible to unforeseen failures. Therefore, the presence of the mechanisms for validation and adaptation of changes at runtime is highly desirable to minimize the presence of errors [BBF09]. Considering the requirements mentioned above, applying model-driven techniques might be beneficial.

2.4.1 Models@run.time in MDE

The classical MDE approach changes the paradigm of creating software from code-centric to model-centric by raising the level of abstraction to implement a system. Therefore, such a paradigm regards models as the central artifacts during the software lifecycle. However, the scope of applying those models is limited to the static time during the design, development, and deployment of commercial products [FS17]. To extend the impact of models, they can also be used to reflect the dynamic semantics of a system for dynamic adaptation and its state validation. The model representing the reflection level of a running system is called models@runtime in the research community [BBF09].

Models@runtime is a promising technique to extend the traditional MDE infrastructure by increasing the role of models from design time to runtime in heterogeneous and distributed

systems. However, the natural complexity of such systems demands particular requirements on the groundwork of models@runtime. These requirements that were collected and analyzed by [Fou+12] are outlined below:

1. Restricted dependency tree. The number of used dependencies and the depth of a dependency tree should be as small as possible. The time to initialize or update a system that uses the models@runtime infrastructure increases proportionally to the increasing size of dependencies.

2. Limited memory footprint. The consumption of main memory determines how demanding a runtime engine is. If the memory footprint is extensive, it might be impossible to run it on small devices. An alternative is to use a lazy loading mechanism to omit running the infrastructure on some nodes. However, this approach also hurts an overall availability of a system since the replication factor lowers.

3. Model variants. Infrastructure should support encapsulation mechanisms for reasoning in side-effects-free environments to avoid any disruptions caused by concurrent modifications of a model.

4. Compatibility with MDE tools. It should be possible to plug MDE tools into the runtime environment. This property aims at fading the boundary between design and runtime. For instance, a node responsible for validating and applying constraints on a runtime model must be able to embed tools for defining those constraints.

2.4.2 Areas of application

MDE models are located above the code's abstraction level and focus on the software lifecycle stages. In contrast, models at runtime are casually connected to the running state of the software. Therefore, depending on the application domain of an operating system, models@runtime can be used in different ways [BBF09]:

- ***Monitoring capabilities.*** Since runtime models reflect the dynamic semantics of a running system, they can be used to monitor the behavior of a managed system or to detect deviation from the specification at an early stage to prevent critical errors leading to the downside of the whole system.

- ***Runtime integration of components.*** Models@runtime could be used to facilitate the dynamic integration of components at runtime. Alternatively, they could advocate during the generation of artifacts and, consequently, embed them into a running system.

- ***Adaptable design.*** Runtime models improve the adaption of design decisions shifting them from static to runtime. Such adaptations could involve the fixes of flaws that could not be anticipated during the initial design stage or add dynamically new requirements.

- ***Autonomous decision-making.*** Through runtime models, adaptation agents could provide strategy techniques by means of metamodels to support more autonomous decision-making of a system.

To summarize, models@runtime is not limited to one concrete domain. Instead, it offers a prosperous scope to tackle various challenges in many research areas. Therefore, apart from the four abstract application areas mentioned above, one should give credit to the following works [BGS19; FS17; Ben+14] where the authors provide more definite disciplines to apply runtime models.

System architecture with models at runtime

A generic architecture proposed by [Ben+14] aims at defining a general system with models@run.time suitable for any domain. This architecture is shown in Figure 2.6.

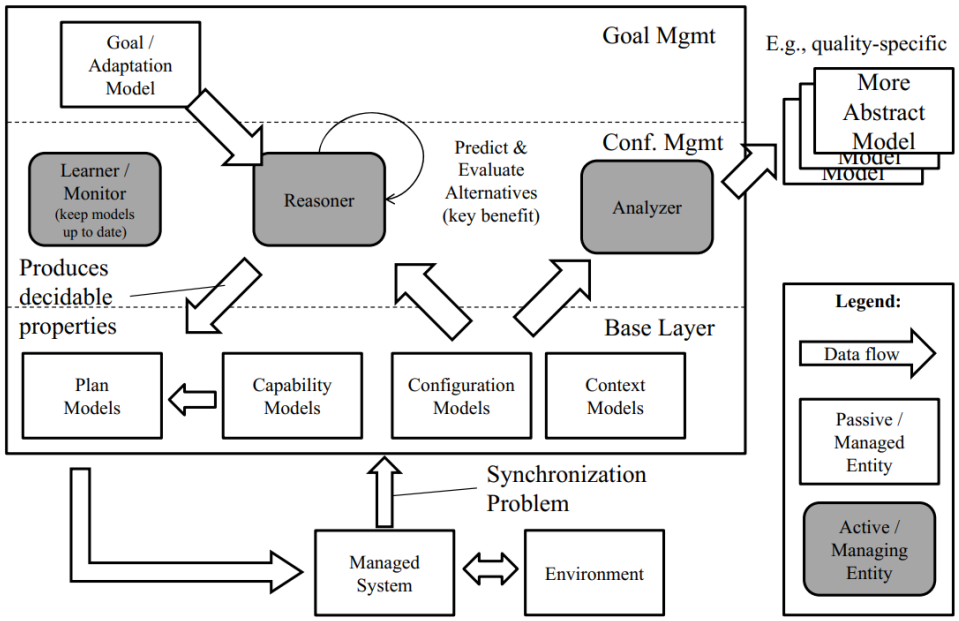

Figure 2.6: Generic architecture for models@run.time systems, taken from [Ben+14]

As can be seen, runtime models do not have direct access to the environment of a system. Instead, they can manipulate the properties via exposed services and sensors on the side of a managed system. Moreover, the models@runtime infrastructure comprises three layers described below.

1. ***Base layer.*** This layer contains models of managed systems. It encapsulates different submodels targeting different goals. *Context models* provide information relevant to some subenvironments. *Configuration models* reflect the current state of the system. Finally, *plan models* serve as prescriptions to guide a system for dynamic adaptation.

2. ***Configuration management layer.*** This layer actively utilizes the models of a base layer to drive the evolution of software. It contains three main subelements. The goal of a *reasoner* is to predict the best actions in the future to achieve the state of software depicted in a target model. The *analyzer* abstracts the base models to decrease the complexity of reasoner tasks and verifies if the state of the system aligns with the goal model. The *learner* helps keep the base models synchronized with the managed system.

3. ***Goal management layer.*** It's a top level of the models@run.time infrastructure that comprises *goal models*. Those models are consumed by a reasoner and define future configurations of a managed system. These models usually change concurrently with changing requirements.

Such infrastructure and its implications are described more formally and in more detail in [Ben+14].

2.4.3 Variant-aware software systems

This subsection should be finished later on based on the work of Karl.

2.5 Constraint and query languages

The broad use and wide acceptance of MDE techniques leverage the software development paradigm by viewing models as central artifacts of the software lifecycle. In this regard, it is vital to guarantee the models' adherence to their formal specifications [CG12].

One of the pure modeling disadvantages is that models cannot fully describe a domain. Models can depict structural semantics via attributes of elements and their interrelations but fail in expressiveness beyond it [WK03]. Figure 2.7 represents the domain model of a project management system. The multiplicity of "manages" between software engineers and projects is unrestricted. Based on it, one can assume that any software engineer can lead any project. However, this relationship might be restricted in reality by the following requirement: ***only senior engineers are allowed to manage projects and managed projects must be started but not yet finished***. Since the graphical form of representation cannot convey such type of requirements it should be defined via textual constraints.

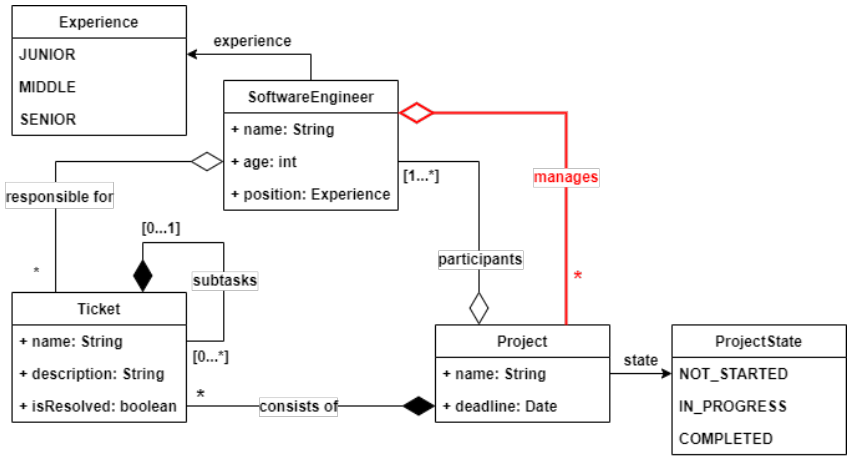

Figure 2.7: Project management system domain model

Domain constraints written using a formally specified language are beneficial to prevent ambiguities while modeling. In this way, the requirements can be precisely interpreted. Nevertheless, textual languages tend to have a steep learning curve and are often repelling with their complexity. On the other hand, modeling graphical tools are intuitive and usually easy to grasp but can often be used only to capture the structural meaning of elements [WK03].

2.5.1 Object Constraint Language (OCL)

Initially, OCL emerged as a supporting tool for UML to mitigate its limitations in the detailed specification of a system. Gradually, its ability to define metamodel, model restrictions, and system requirements extended the usage of OCL. Currently, OCL is an adopted standard of OMG to specify, transform, and constraint models in MDA [CG12].

The essential properties of OCL are that it is a typed and side-effects-free language [WK03]. The former implies that the execution result of a query must always resolve to some type. The latter means that a query execution does not alter the state of constrained object nets.

OCL has rich application scope and can define the following constraint categories [CG12]:

- invariant restrictions on a class level that must be satisfied with every instantiation from a constrained classifier
- rules which every derived field should confront while being constructed

- pre- and post-specifications, which must remain true before and, respectively, after some operation takes place
- query rules to traverse a model and return information to a requester

OCL allows navigating through classes, their attributes, and their relations among them denoted by associations. Depending on the target type of an element, navigation results either in a concrete value or a collection.

Constraints compared to queries

The question that might arise is whether a constraint differs from a query and whether those concepts can be used interchangeably. A constraint is a logical restriction imposed on a property of a model element. Evaluating a constraint means reasoning whether a restricted property holds true or false for any concrete instance of a constrained model element. If a domain model specification restricts a model, such a model can be considered valid if all instances are instantiated within constraint boundaries [WK03].

On the other hand, a query is a request to retrieve information from a targetted environment [Rei81]. In the context of modeling, the environment for querying would be a created model. The query command that returns a boolean type can be considered both a query itself and a constraint [WK03]. In order to illustrate the interchangeability of those two concepts, the above-defined constraint is implemented in two ways: with OCL and the query language for graphs and web ontologies SPARQL.

```
context <SoftwareEngineer>
inv: self.manages -> notEmpty() implies
        self.position = Experience.SENIOR AND
        self.manages(project | project.state = ProjectState.IN_PROGRESS
```

Figure 2.8: OCL constraint for the project management system

```
PREFIX software_engineer <http://sample-management-system.de/software_engineer>
PREFIX project_state <http://sample-management-system.de/project_state>
PREFIX experience <http://sample-management-system.de/experience>

ASK { // --> ASK operator returns true on a pattern match.
        ?engineer software_engineer:name ?name ;
                software_engineer:position ?position ;
                software_engineer:manages project_state:IN_PROGRESS .
        FILTER (?name = <provided_identifying_name>) .
        FILTER (?position = experience:SENIOR) .
}
// true results in constraint conformance;
// false results in a constraint violation
```

Figure 2.9: SPARQL constraint for the project management system

2.5.2 Dynamic constraint at runtime

The recent approaches and techniques [SZ16; FS17; BBF09; KRS15] to extend the application scope of models to runtime raise a challenge of their consistency in the presence of

dynamic changes. Hence, the definition and co-evolution of integrity constraints alongside a model at runtime is an ongoing research objection.

Constraint languages play a primary role in the MDE toolkit to express the additional domain requirements supplementing classical modeling strategies during the software lifecycle. However, despite being powerful and expressive by definition, one of their limitations at runtime is the inability to adapt to the continuously changing environment automatically [SZ16].

Runtime models must also provide means to capture, evaluate, and evolve constraints at runtime clearly and intuitively for end-users. They differ from traditional static MDE ones by automatically triggering the evaluation engine upon creating, updating, or removing model elements [KRS15].

The unique trait of runtime models to evolve at runtime alongside a managed system is a double-edged sword. On the one side, it enables the dynamic adaptation of integration of new components without hurting or downgrading availability requirements. However, on the other side, even a trivial change in a model might lead to a complete invalidation of constraint sets or, even worse, result in false positive/negative results [Vie+21]. Therefore, there is an urge for promising techniques that could be used at runtime to adapt constraints during the model evolution dynamically.

The research paper of [Cur+10] focuses on updating integrity constraints on a database schema and supporting legacy query evaluation in the face of schema evolution. The main two components are Schema Modification Operators (SMO) and Integrity Constraints Modification Operators (ICMOs). The former contains the current state of the schema with the listed operations that it underwent. The latter serves to save all integrity adaptations in sequential order. Furthermore, the authors propose a semi-automated engine that uses SMO and ICMO objects to recreate the elder schema version and to provide backward compatibility for legacy queries and operators. As a result, this approach allows a schema evolution where old queries can be executed on a more contemporary schema snapshot.

Another author [SS09] implemented a framework that detects changes in a model and compensates for any inconsistencies. The engine continuously performs the model analysis in the background. Upon encountering modified elements during analysis, the engine notifies an end-user with the list of affected model entities and actions to bring the model back into a uniform state. The end-user is then responsible for approving or discarding the suggested steps of an evaluator.

The next approach [GRE10] tackles the problem of runtime model evolution by introducing a mechanism for verifying models and metamodels in a language-agnostic way. Moreover, it maintains the correctness of constraints during the model evolution. The main element is a model profiler that tracks the state of all model elements mapped to a constraint. The set of models covered by a particular constraint is called a scope. Whenever a change happens in a model, every affected scope is reconfigured independently. The proposed algorithm is presented in figure 2.10 below.

```
processModelChange(changedElement)
if changedElement was created
 for every definition d where type(d.contextElement)=type(changedElement)
   constraint = new <d, changedElement>
   evaluate constraint
else if changedElement was deleted
 for every constraint where constraint.contextElement=changedElement
   destroy <constraint, changedElement>
for every constraint where constraint.scope contains changedElement
 evaluate <constraint, changedElement>
```

Figure 2.10: Algorithm for model change adaptation at runtime, taken from [GRE10]

Finally, the architecture offered by [AGK12] focuses on providing consistency for multilevel models. Since multilevel models are level-agnostic and all are treated uniformly, it is possible

to change the entity hierarchies at runtime. Nevertheless, such dynamism also implies subsequent complexity because changes on higher levels may considerably impact lower-level models. For this reason, there is a demand for consistency checking mechanisms to keep the model conforming to the specification. A proposed architecture comprises three elements - deep models, an emendation service, and an impact analyzer.

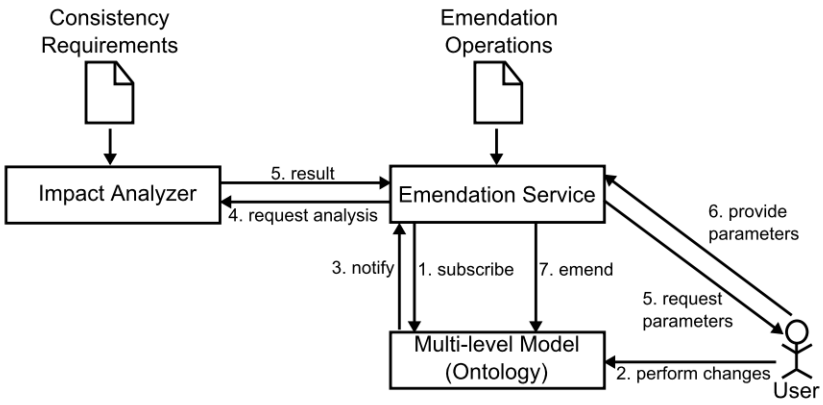

Figure 2.11: Emendation service architecture, taken from [AGK12]

As shown in figure 2.11, the integrity and on-the-fly correction of inconsistency are performed in the following steps:

1. *Emendation service* subscribes to a *deep model* and listens for changes
2. The *emendation service* is notified with every model change and forwards the request to the *impact analyzer*
3. The *impact analyzer* determines the affected by the change model elements
4. The *emendation service* generates configuration instructions and forwards them to a user
5. A user selects or aborts offered configuration steps
6. Finally, the *emendation service* performs the actions approved by a user

To summarize, the prerequisite to enforcing the consistency and validity of a model in the presence of changes is a long-running research question raised and tackled by various research communities. However, the common property of the above methods concerning the constraints and model co-evolution is that such a system needs two key elements. Firstly, there should be an analyzer to observe changes in a runtime model. Secondly, a validation service should be responsible for adapting constraints at runtime by having a connection link with an end-user.

3 Concept and requirements

This chapter presents the main prerequisites of a system that supports constraint definition, evaluation, and modeling in the face of runtime changes and entity-relational model variability. At first, we define the necessary types of constraints by end-users. Then, based on the analysis, the primary requirements for constraint definition and its modeling means are defined. Finally, the key aspects are summarized, which serve as the foundation for future design and implementation.

3.1 Reasoning about constraints by example

The initial objective of this chapter is to reason about potentially suitable types of constraints that end-users could impose. For this reason, the domain model of a project management system mentioned in the previous chapters is reintroduced. However, the simplicity of such a model limits the scope of potentially required constraints. Hence, the domain model has been enriched with new domain elements, relations, and attributes. The refined domain model is shown in figure 3.1 and is accompanied by the explanation below. Furthermore, it is worth noting that the following model claims to be partially defined and exhaustive. The main focus here is on potential constraints that could increase its expressiveness.

- ***Client*** - a person that requests the development of a new software product. Depending on the purchased status, the client is provided with an appropriate package of services.
- ***Client status*** - two types are available. The difference between them is that the *premium status* permits demanding additional requirements compared to the restricted *basic status*.
- ***Project*** - a simplified entity of a managed project in the real world. Every project has a unique name and deadline. Software engineers develop the project, and its lifecycle consists of numerous sprints.
- ***Project state*** - an enumeration class to reflect the current status of a project.
- ***Sprint*** - a short time-limited range of time within which a team should complete the allocated amount of work. Every sprint consists of tickets and developers taking part in it.
- ***Ticket*** - a concrete unit of work that a software engineer must do. It contains the name, description, and progress status. Each ticket can be assigned to only one software engineer.

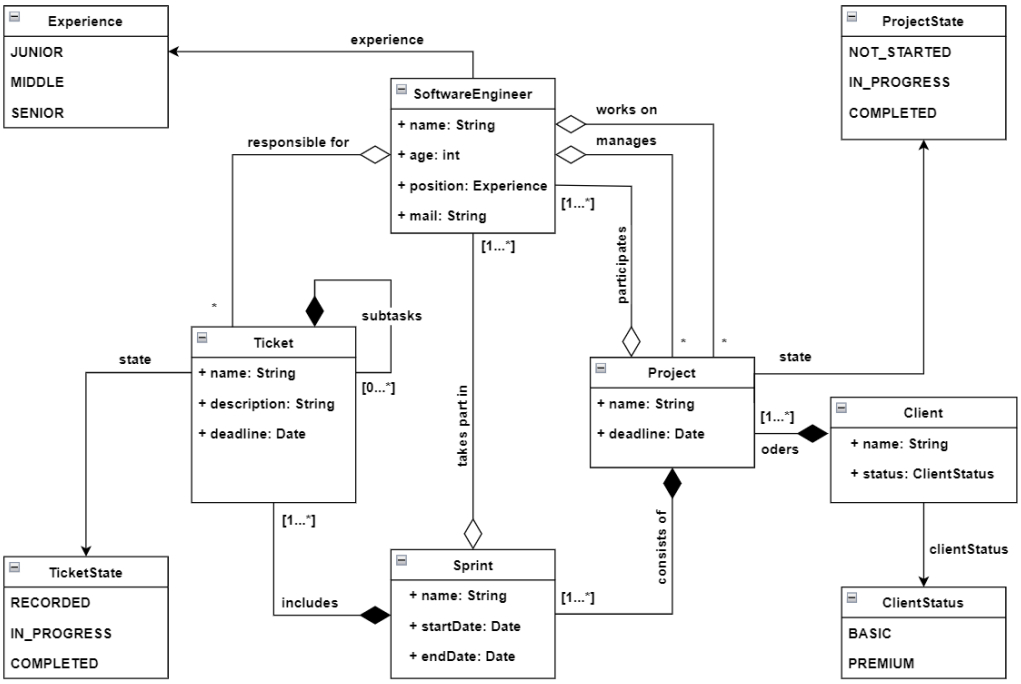

Figure 3.1: Elaborated domain model of a project management system

- ***Ticket status*** - an enumeration class to concretize the state of a ticket. *Recorded* stands for the creation of a ticket. *In progress* implies the fact that the ticket was assigned to a software engineer. *Completed* highlights the full and correct implementation of requirements within a ticket. Finally, large tasks can be decomposed into subtasks to facilitate development.
- ***Software engineer*** - an individual responsible for the progress of a project. They participate in sprints and are accountable for the implementation of current tasks. Depending on the experience, there is also an opportunity to manage projects.
- ***Experience*** - an enumeration class to reflect the proficiency level of a software engineer. In order to stay concise, only technical labels are considered, and the management roles are excluded.

As concluded in the foundation chapter, modeling diagrams are only sufficient to specify the structural requirements of a domain. By looking only at the defined model, some questions might already arise regarding the functionality of a system that uses this entity diagram at its domain level. For instance, as of right now, it is already unclear what benefits bring a premium client status or what conditions should be met by a software engineer to be able to manage a project. Imposing additional constraints would allow applying additional prerequisites for a domain model.

Having characterized the structural facet of a system, it is time to take the role of an end-user and analyze what other requirements would be necessary for such a system. The concrete constraints defined in table 3.1 will serve as the foundation for further constraint classifications in the following subchapter.

As observed, all specified constraints can be grouped into three different categories depending on what properties of a model element they intend to restrict. Constraining attributes and associations can be regarded as two independent categories, whereas more complex constraints require aggregating both types. Nevertheless, such overarching and general groups should be split further into more precise constraint specifications in order to be explicitly formulated by end-users.

№	Requirement	Related to
1	Name of software engineers must be unique	Attributes
2	Deadline for a ticket must be set first before assigning a responsible	Attributes
3	Closed task must have a deadline	Attributes
4	Software engineer must be adult	Attributes
5	Software engineer's age can be set only in numeric format	Attributes
6	Name of an employee must consist of name and surname separated by space	Attributes
7	Email address must contain a company domain	Attributes
8	Length of an employee name must not exceed 255 characters	Attributes
9	Technical lead must possess at least senior development skills	Attributes
10	Names of all tasks assigned to a software engineer must be unique	Attributes and associations
11	All tickets included in a sprint must have either name or description	Attributes and associations
12	Technical lead can manage concurrently at most two project	Attributes and associations
13	Completed project cannot have opened tasks	Attributes and associations
14	Task cannot be a subtask itself	Attributes and associations
15	All managed projects must be started and not yet finished	Attributes and associations
16	Software engineer can have at most fifteen assigned tickets in total, and no more than five opened tickets in every project	Attributes and associations
17	Number of tasks including their subtasks in a new sprint cannot be more than the average task number within the last three sprints	Attributes and associations
18	Maximum age of all participants in a project must be sixty-five	Attributes and associations
19	Premium clients can require no more than twenty-five percent of all senior engineers to be involved in a project, whereas basic clients only 10 percent	Attributes and associations

Table 3.1: Project management system: non-structural requirements

Because of the infinite number of requirements that an end-user might specify for a domain, one of the consequences is that some of them might be missed in this section, and, subsequently, some types of constraints might be omitted. Therefore, one of the requirements for the future framework that supports the specification and validation of constraints at runtime is the ability of an end-user to extend the scope of possible constraint specifications by defining them at runtime.

3.2 Constraint types specification

Figure 3.2 presents the decomposed and more concrete classification of constraint types. It is based on the defined end-user domain requirements for a project management system and official documentation of well-established constraint languages [CG12; PLS14; Par+20].

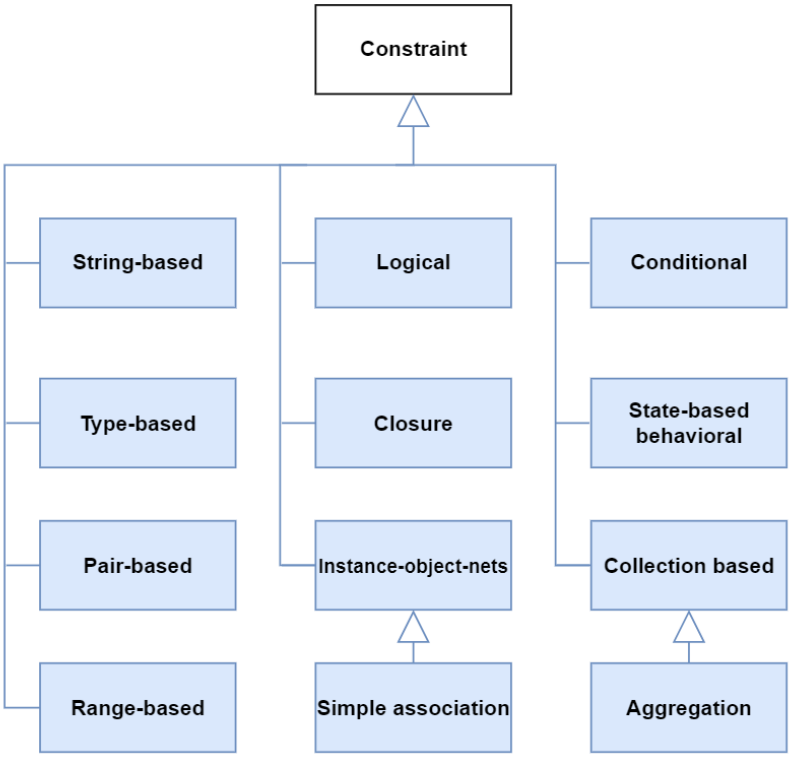

Figure 3.2: Constraint types specification

Below every constraint classification is described in more detail. Moreover, the connection between a constraint category and its relevance to specifying above mentioned requirements for the project management system is demonstrated.

1. String-based constraint - specifies restrictions applied to attributes with a string representation.
   a) *Maximum/Minimum length* - enforces an attribute's maximum or minimum limit length to a specified value.

      Relevant for requirements: №2, №3, №8, №11

   b) *Pattern match* - enforces an attribute to adhere to a specified regular expression. This constraint helps satisfy the internal policy standards of a certain environment.

      Relevant for requirements: №6, №7

c) ***Unique*** - imposes an attribute value's uniqueness within all instances of a certain model element.

Relevant for requiements: №1

2. Type-based constraint - enforces the correctness of an attribute's data type. Unfortunately, some environments do not provide strict type checking. Implementing type-checking mechanisms directly in source code might be costly or even impossible. In this case, one could leverage strict data type enforcement to the constraint engine to avoid errors and mistakes during the software development process.

   Relevant for requirements: №5

3. Pair-based constraint - defines conditions to be satisfied between two attributes. The attributes might not be limited to only one model element. In this respect, the attributes relevant to different model elements can be compared via navigation through an instance model or a data graph.

   a) ***Equals*** - constrains two attribute values to be of the same value.

   Relevant for requirements: №3, №9, №12, №13, №15, №19

   b) ***Mismatched*** - constrains two attributes to be of distinct values.

   Relevant for requirements: №14

   c) ***Less/More than*** - enforces that one attribute value must be strictly less or more than the second attribute value.

   Relevant for requirements: №16, №19

   d) ***Less/More than or equal*** - enforces that one attribute value must be either less or more than the second attribute value or be equal to the second value.

   Relevant for requirements: №14, №17

4. Range-based constraint - limits valid values of a single attribute to be within the specified range depending on the operation.

   a) ***Less/More than*** - enforces the value of an attribute to be within a specific range, excluding boundary values.

   Relevant for requirements: №4, №18

   b) ***Less/More than or equal*** - enforces the value of an attribute to be within a specific range, including boundary values.

   Relevant for requirements: №8, №12, №16

5. Closure constraint - returns the result of direct attribute elements, elements of its attribute elements, and so forth until it reaches the leaf level. This operation helps analyze the transitive dependencies of an instance element.

   Relevant for requirements: №14, №17

6. Instance-object-nets constraint - defines a constraint that spans multiple instances.

   a) ***Maximum/Minimum cardinality*** - specifies the maximum or the minimum number of elements a target instance can be associated with.

   Relevant for requirements: №13, №16, №19

7. Simple association constraint - a subclass of instance-object-nets constraints that target constraints spanning a modeling element's direct neighbors. This category is treated

separately since the integrity and synchronization mechanisms must not traverse a deep instance hierarchy in the presence of runtime changes.

Relevant for requirements: №10, №11, №12, №14, №15, №17, №18

8. State-based behavioral constraint - specifies the instance evolution process when a transition of one instance attribute to another state is only possible if the state of another attribute is satisfied by a given requirement.

   Relevant for requirements: №13

9. Conditional constraint - enforces the evaluation engine to trigger the correspondence of an instance to a constraint if and only if a given condition is satisfied. The optional alternative path can also be provided in case of non-fulfillment of the condition.

   Relevant for requirements: №2, №3, №12, №13, №19

10. Collection-based constraint - defines boolean operations to be satisfied for direct attributes of collection data type or navigation ends that result in a collection.
    a) *Unique* - enforces the unique value of each attribute in a collection.

       Relevant for requirements: №10

    b) *Is/Not empty* - checks whether a collection contains or does not contain elements.

       Relevant for requirement: №15

    c) *Match any/all/none* - checks whether any/all/none elements of a collection satisfy a given requirement.

       Relevant for requirements: №13

11. Aggregation constraint - a subcategory of collection-based constraints to group the values of an array into a single value depending on the operation.
    a) *Maximum/Minimum value* - checks whether a collection's maximum/minimum value is within the provided range.

       Relevant for requirements: №18

    b) *Average value* - checks whether the average value of a collection is within the provided range.

       Relevant for requirements: №17

    c) *Total count* - checks whether the total number of elements satisfies the required number.

       Relevant for requirements: №16

    d) *Distinct total count* - checks whether the total number of unique elements if a collection satisfies the required number.

       Relevant for requirements: №16, №17

12. Logical constraint - constraint consisting of several nested constraints grouped by a logical operator. The overall constraint conformance is evaluated among all nested constraints, and its result depends on a logical operation.
    a) *And* - the constraint is satisfied if all nested requirements are satisfied.

       Relevant for requirements: №16

    b) *Or* - the constraint is satisfied if at least one of the nested requirements is satisfied.

       Relevant for requirements: №11

Requirement №	Formulating via constraint types
№1	String-based
№2	Conditional, string-based
№3	Conditional, string-based, pair-based
№4	Range-based
№5	Type-based
№6	String-based
№7	String-based
№8	Range-based, string-based
№9	Pair-based
№10	Simple association, collection-based
№11	Simple association, logical, string-based
№12	Simple association, pair-based, range-based, conditional
№13	State-based-behavioral, conditional, pair-based, instance-object-nets, collection-based
№14	Simple association, closure, pair-based
№15	Simple association, collection-based, pair-based
№16	Instance-object-nets, range-based, aggregation, logical
№17	Simple association, closure, aggregation, pair-based
№18	Simple association, aggregation, range-based
№19	Instance-object-nets, conditional, pair-based

Table 3.2: Relation between requirements and constraint classifications

Considering all mentioned above, table 3.2 depicts how every requirement defined for a sample project management system can be formulated via the defined constraint classifications.

To sum everything up, this subsection presented the minimal number of constraint types to be defined by an end-user that the final framework should support. One point to highlight is that depending on the defined domain and possible requirements, there might be a need for more concrete or specialized constraint types. Therefore, in order to provide a flexible constraint definition, the framework should support the functionality to expand the scope of constraints at runtime.

3.3 Constraining variant-aware entity model at runtime

In addition to having defined minimum types of constraints that a constraint engine should be able to support and evaluate, it is vital to consider the high dynamism of runtime models. Therefore, to clearly state the requirements of a system, it is necessary to view the problem space from several angles and stick to the following questions:

1. What should be the behavior of a system to keep constraint integrity in the face of a runtime model evolution?
2. At what level of modeling should constraints be defined and evaluated?
3. What elements and actions should comprise a constraint life cycle?
4. How do changes at the instance level affect the domain space of an end-user, and what actions are required to bring object nets into a valid state in case of violations?
5. How do different types of model changes at runtime affect the validity of constraints, and what measures should be taken to guarantee model constraint correctness?

6. How and to what extent should a logical constraint reasoner framework be embedded into a user-driven constraint modeling system?
7. How should constraints be constructed by end-users in a user-driven constraint modeling system?

Taking into account the questions raised above, the following subchapters elaborate on the aspects of system behavior at distinct levels of interaction with a focus on constraint integrity at runtime.

3.3.1 Variant-aware application with dynamic constraints. Abstract view.

Figure 3.3 presents the most abstract view of system interactions. All the defined activities happen at runtime. Thus, such architecture eliminates the burden of redeployment on adding new functionality. At first, an end-user depicts structural prerequisites for a system by creating an entity relationship diagram in the scope of a targeted domain (*Create domain*). In the next step, additional requirements are formulated by constraining the predefined environment (*Create constraints*). Before the model utilization, a reasoner must evaluate the constraints to check the fulfillment of rules (*Check constraints*). The model is then used eventually by stakeholders until one of two actions occurs (*Use domain model*):

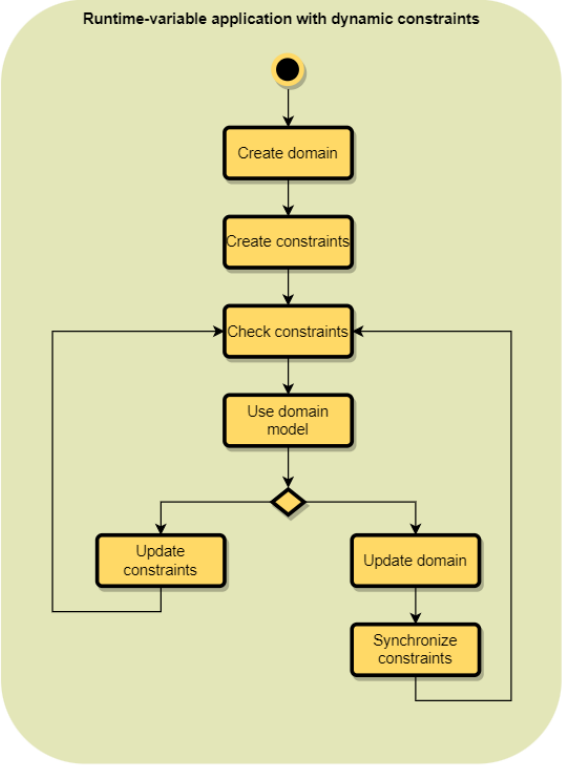

Figure 3.3: Abstract runtime model lifecycle with constraint support

1. There is a new requirement that affects the functionality of the domain. Metting such a requirement results in updating the domain constraints (*Update constraints*) and requires reevaluating affected rules (*Check constraints*).
2. The second possible scenario is the emergence of new structural requirements that lead to changes in the domain itself (*Update domain*). In this case, there is a probability that

some of the constraints defined earlier will become irrelevant. In this regard, the process of synchronizing constraints (*Synchronize constraints*) and checking their correctness (*Check constraints*) are necessary.

These steps repeat cyclically and happen during system operation. Further activity diagram focus on constraint integrity and its life cycle at runtime in more detail.

3.3.2 Constraint life-cycle

Figure 3.4 showcases the use cases of applying constraints on a model at runtime. There should be three explicit operations that an end-user can perform regarding limiting a domain. Namely, a user must be able to create, delete, and update constraints. Below, each of the available operations is thoroughly defined.

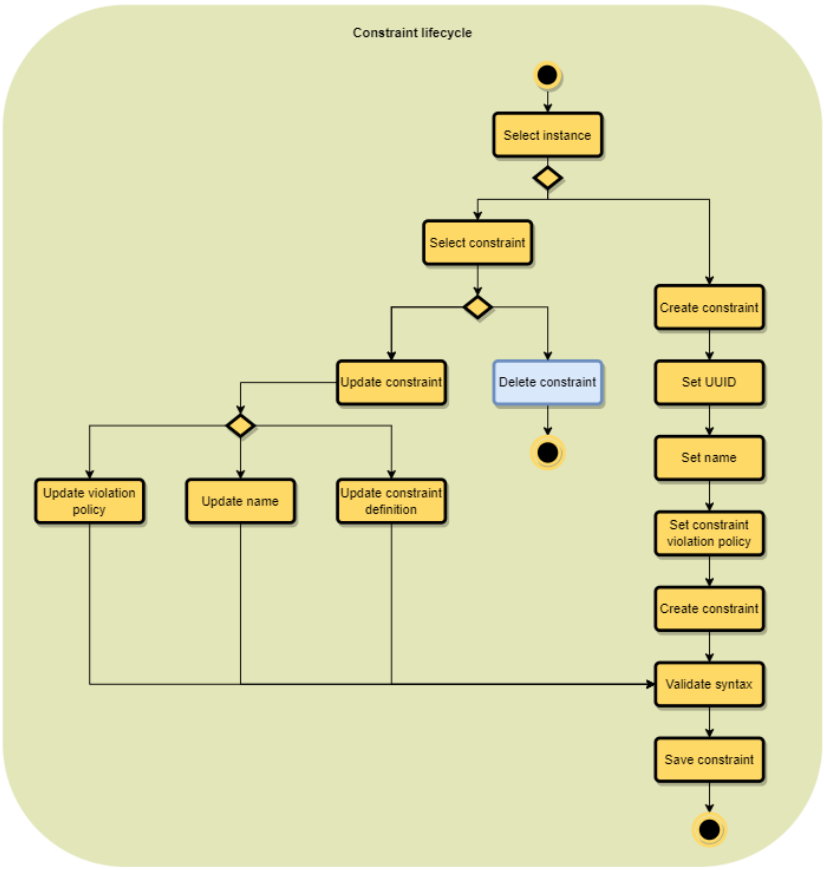

Figure 3.4: Constraint life cycle

- ***Create constraint*** A constraint is regarded as being successfully created if it has four components:
  1. *Unique identifier* - for internal identification and verification of a constraint
  2. *Name* - for various notifications related to the created constraint
  3. *Violation policy* - violation of some constraints might be fatal. In contrast, some other restrictions should not hinder the integrity mechanisms of the whole model. Therefore, to classify constraints by the severity of their violation, this parameter must be set upon every creation

4. *Constraint* - a semantic definition of a requirement to restrict a domain. It must adhere to the valid syntax of a constraint engine and be syntactically correct.

- ***Update constraint*** - includes updating its name, violation policy, or definition itself. The refined constraint must be checked against its syntax validity regardless of an edited attribute.
- ***Delete constraint*** - this operation does not require any particular actions. The process to purge the existing constraint from a persistent unit should be leveraged to the component where all constraints are stored.

Finally, the persistence layer must preserve all performed actions or changes independent of the path taken by an end-user.

3.3.3 Model instances and constraints at runtime

This chapter reveals an answer to providing model integrity and constraint enforcement during the usage of a model by end-users at runtime. As depicted in Figure 3.5, depending on the instantiation operation, the system must perform various activities that, in the end, converge to the validation of the constraint's integrity, analysis of affected indirect constraint, and persisting changes if all the defined restricting rules are satisfied.

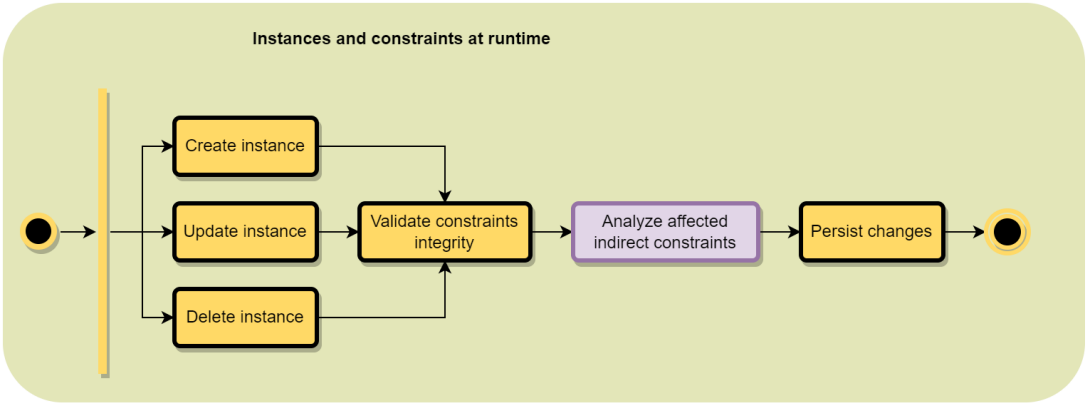

Figure 3.5: Model instantiation and constraints integrity. Abstract view.

The *Analyze affected indirect constraints* step is colored differently to highlight the necessity in terms of enforcing the integrity of non-structural requirements. It is worth mentioning that this step is relevant if and only if, in the whole model space, there exists at least one constraint spanning several object-nets instances that refers to this type of instance via the means of navigation. Figure 3.6 depicts a snippet of an instance space from the project management system to clarify this statement. As such, two software engineers are employed in the CMS project. The company rules require that every project must have at least 30 percent of junior developers among all involved software engineers to encourage the training of young specialists. As can be seen in the current form, this requirement is satisfied because John is working on this project, and the current ratio of junior developers is fifty percent. However, depending on the situation, one of the following might happen:

1. **Create-instance-relevant.** Two experienced middle developers joined the project. Therefore, the constraint defined in the scope of a project does not hold anymore because the ratio of juniors is now only twenty-five percent.

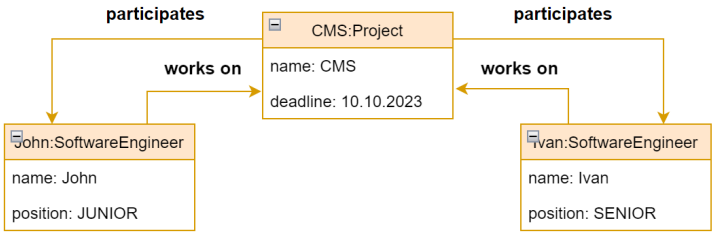

Figure 3.6: Project management system. Instance space.

2. **Update-instance-relevant.** Time passes, and our specialist John has become a middle developer. But, again, the update of proficiency of John invalidated the requirement applied to a project.

3. **Delete-instance-relevant.** After some time, John decides to leave the company, and therefore the constraint turns out to be violated again.

It can be observed that changing the shape of one instance affects the validity of object-nets constraints imposed on other modeling elements. In this regard, there must exist a mechanism that detects the violation of domain-applied rules caused by updating indirect instances. One of the following approaches could be viable to tolerate such a scenario:

1. *Synchronous thorough constraint evaluation on every instance-related operation.* Before persisting any changes, the main thread is blocked until all constraints in the system's scope are reevaluated. The obvious downside here is a considerable performance and useability overhead because the required time incrementally increases with the number of elements and defined constraints.

2. *Asynchronous thorough constraint evaluation on every instance-related operation.* This mechanism is analogous to the one defined above but happens in the background transparently to an end-user. All found constraint violations are delivered to users upon successful reevaluation via notifications. Even though this approach does not harm usability, it still triggers significant overhead on the system.

3. *Possessing support for backward-related links.* This strategy extends the scope of constraints by linking the constraints of one model element to all the other elements that are part of a constraint. In other words, every module element X0 has links to all model elements X1...Xn that has defined constraints, including X0. Considering the example above, since the constraint in the context of "Project" has ties to any instance of the type of "SoftwareEngineer", every instance of "SoftwareEngineer" will maintain links to this constraint. Therefore, every instance-related procedure in the scope of "SoftwareEngineer" will trigger the evaluation of only linked constraints in its scope. This approach eliminates the need for complete constraint assessment but increases the evaluation engine's complexity.

Since this issue has multiple ways to be resolved and is subject to further theoretical considerations, no explicit solution is provided here.

Instantiation and constraint validity enforcement

Before creating a model element instance, the system must perform several substeps to guarantee the overall model integrity and satisfiability to non-structural requirements. The upcoming reasoning is based on figure 3.7. First, the respective classifier must be fetched to

create an instance (*Fetch classifier*). The classifier contains necessary information about defined attributes and connection links to other elements. Apart from it, every classifier must have constraints encapsulated (*Fetch constraints for classifier*). As the next step, object instantiation takes place (*Initialize instance*). Here, an instance is populated with concrete values defined by an end-user or with predefined fixed values of a classifier and its supertypes. After that, the reasoner evaluates whether an instance meets all the specified constraints (*Validate constraints*). The initialization of evaluation must be transparent for a user and should be initiated on the user's intention to create an instance. If all restriction rules are valid, no further actions are required from a user. On the other hand, if at least one of the constraints with a strict violation policy is disregarded, the user must manually edit the instance to conform to the rules (*Edit instance*). Finally, an instance is created upon adherence to all the constraints (*Create instance*).

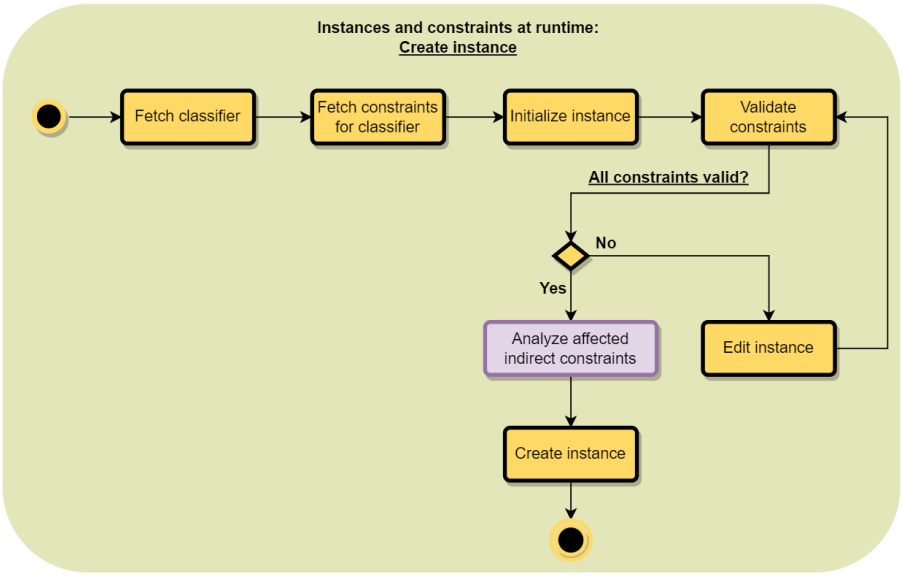

Figure 3.7: Model instantiation and constraints integrity. Create instance.

Updating instance and validating constraints

Updating an instance is similar to instantiation and overlaps in many steps. As shown in Figure 3.8, a classifier of an instance must be obtained first (*Fetch classifier*). Then, all the relevant constraints are extracted from the classifier (*Fetch constraints for classifier*). The next step is to query the existing instance to be updated (*Fetch instance*). The following operations must be available in the scope of an instance update: updating attribute values (*Update attribute value*), association addition (*Add associated instance*), and association removal (*Remove associated instance*). After updating an instance, it is evaluated on the conformance of all constraints (*Validate constraints*). Again, this step must be transparent for an end-user in the sense that it does not require any additional interactions from the side of a user. In case of strict constraint violations, a user must revise an instance until all non-functional requirements are satisfied (*Edit instance*). Finally, the updated instance must be persisted (*Update instance*).

Deleting instance and providing constraints integrity

Deletion operation does not require any additional steps to provide integrity for constraints specified in the scope of the instance. Therefore, as depicted in figure 3.9 the process of deleting an instance is relatively straightforward. Nevertheless, the analysis step of potential

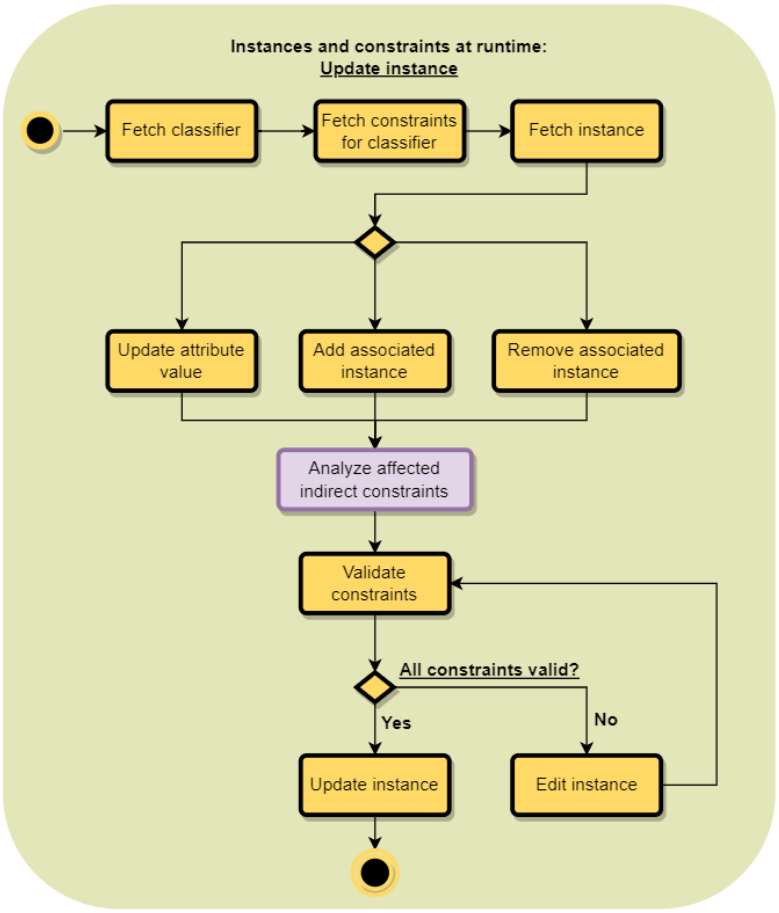

Figure 3.8: Model instantiation and constraints integrity. Update instance.

indirect constraint violations (*Analyze affected indirect constraints*) must occur if this instance is part of any restriction rules defined in the context of other model elements. After this step, the instance is removed from the system, and the changes are persisted (*Remove instance*).

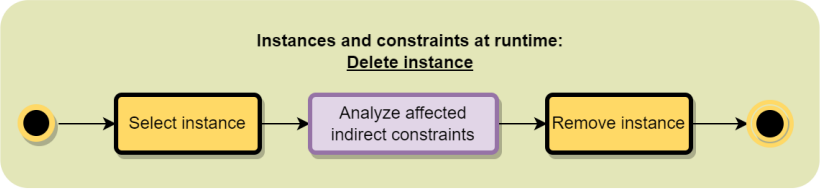

Figure 3.9: Model instantiation and constraints integrity. Delete instance.

3.3.4 Model evolution and constraints at runtime

The previous section focused on specifying requirements for constraint integrity in the face of instance-related operations. This section addresses the problem of maintaining constraints' correctness during the domain model evolution. Every change in the model space might affect the correctness of already defined non-structural requirements. For instance, removing a classifier might invalidate all constraints that refer to it because it does not exist anymore. Such an operation would require a manual end-user intervention to bring the constraint scope again into a consistent state. On the other hand, invalid constraints caused by updating some parts of a domain could be adapted automatically and stay transparent for end-users. To reason

about the necessary means to provide constraint adaptability at runtime, this section presents several activity diagrams that tackle each possible domain change and offers mechanisms for constraint's integrity.

Figure 3.10 shows the most general overview of actions that constitute the evolution of a domain. The transition of the domain from one state to another might be triggered by changes applied to the whole model element or exclusively to its attributes and associations. Regardless of the type of change, adding new parts to the model does not invalidate already defined constraints because, when those restrictions were specified, those new elements were not considered. However, any update or delete operation has the danger of violating the properness of one or several constraints whose definitions refer to a concerned model element. For this reason, before persisting any changes at runtime with the domain, the system must determine the set of affected constraints and offer ways for their synchronization to restore their correctness. In the diagram, this step is called *"Synchronize constraints"* and serves as an abstraction for the actions that vary depending on the model evolution. Figures X-Z elaborate on it in more detail by explaining how constraint integrity is provided for every type of domain change at runtime.

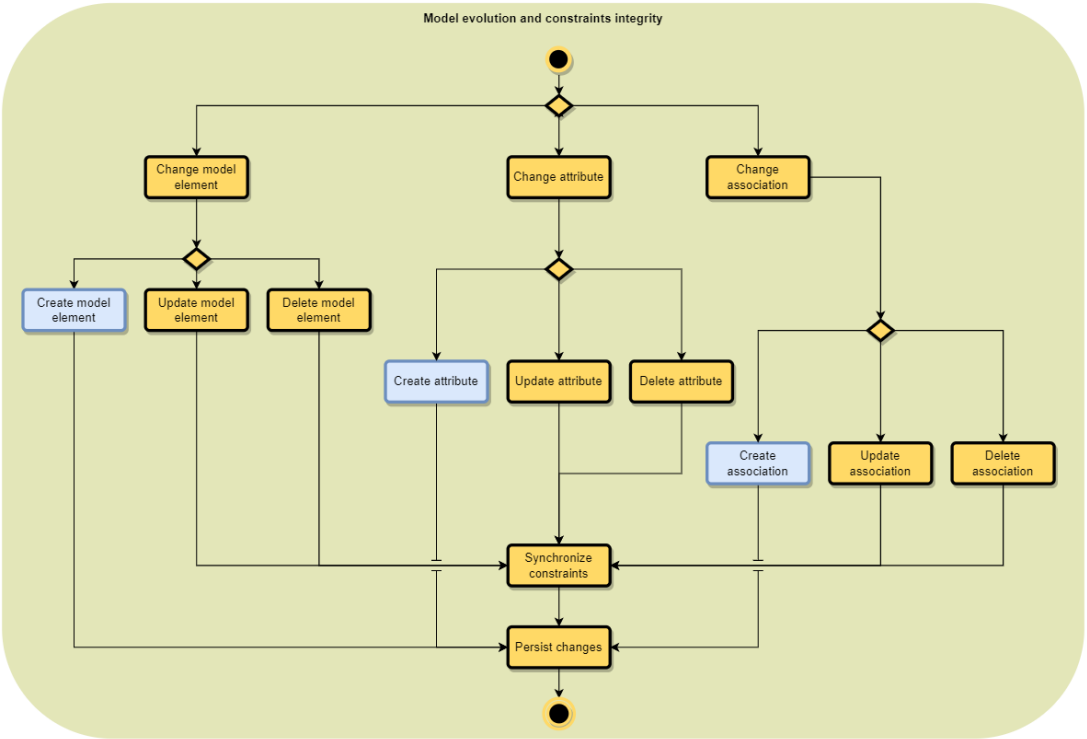

Figure 3.10: Model evolution and constraints integrity. General overview.

Model element evolution and constraints at runtime

Figure 3.11 depicts the required measures a system must undergo to ensure the correctness of specified constraints in the scope of a model element. The *"Create model element"* step does not violate the range of already created constraints. Therefore, the changes might be persisted straight away. On the other hand, updating a model element requires additional steps from a reasoner engine to analyze invalid constraints and bring them into a consistent state. This operation might be of two possible types, namely *Update attribute* or *Update association*. The mechanisms for constraint synchronization are explained below for every kind of potentially updated element.

Nevertheless, deleting an element is relevant only for the scope of a single model element and is explained in this section. This operation requires obtaining the list of all constraints defined directly for this instance that will be removed together with the component (*Fetch constraints to be removed*). Next, the constraint engine must analyze and discover all the constraints that will be invalidated after the deletion of this domain unit (*Fetch affected constraints*). Providing the means of automatically reconfiguring affected constraints is not part of this work due to the rich research scope of a possible solution. Therefore, the process of updating all the invalid rules should be leveraged to an end-user. In turn, the user can either abort the domain unit deletion, edit the affected indirect constraints manually, or remove them from the system. Depending on the further selected activity, the changes may be persisted.

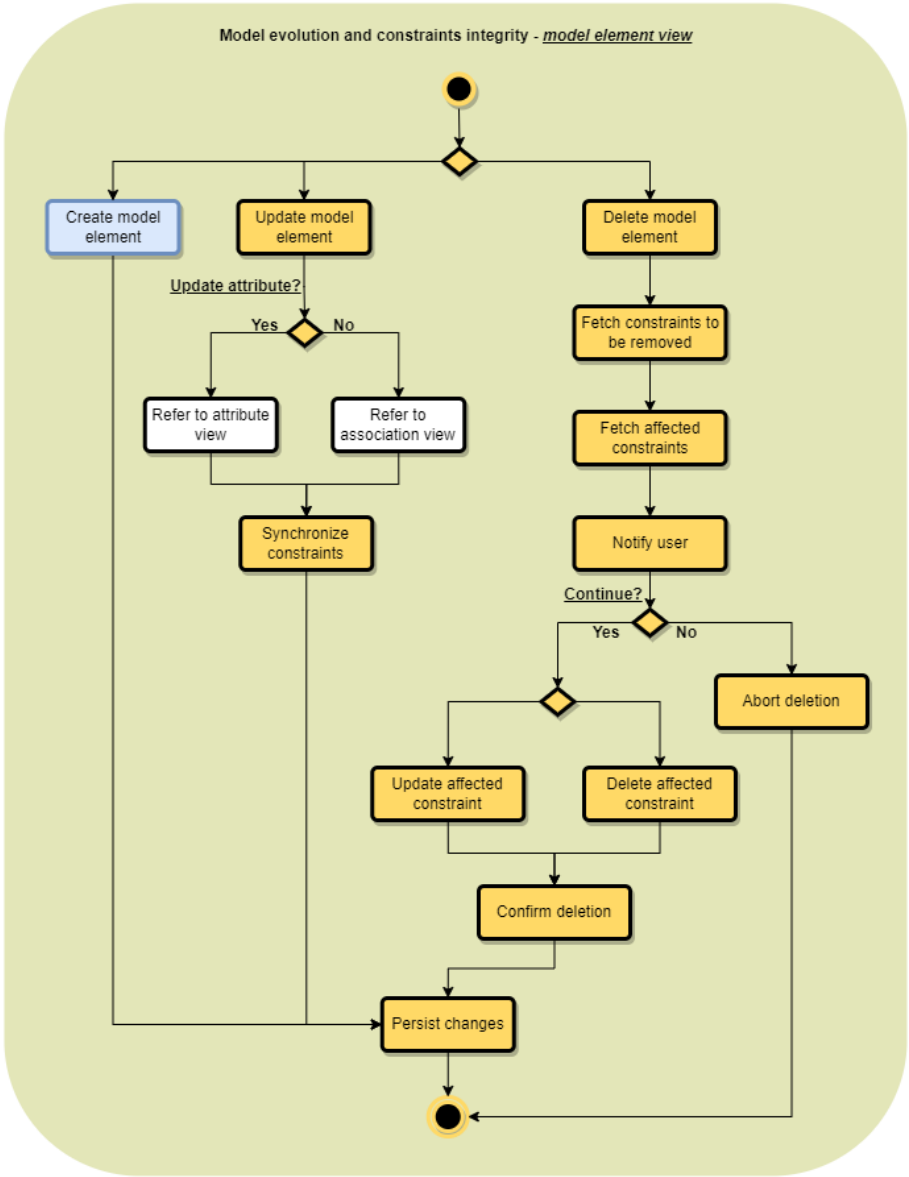

Figure 3.11: Model element evolution and constraints integrity.

Attribute changes and constraints at runtime

Figure 3.12 clarifies how a system must behave in case of an attribute-related change in one of the domain elements. Creating a new attribute does not require any additional means of delivering the integrity of constraints. Therefore, an update on a model element must be saved

directly without further stages. Updating an already existing attribute requires the revision of all affected restrictions. Suppose such an operation involves an update of the name or the type of an attribute. In that case, the system must be able to correct the references in all constraints automatically. If the system fails to actualize the affected rules or some error occurs, an end-user must be notified about the cause of the failure. Deleting an attribute also involves several steps, but in comparison to the update, this operation is meant to happen with the user interaction. At first, the system must collect all the constraints in the scope of the attribute to be deleted. Then, these constraints must be deleted as well. In addition, the system must analyze the domain space and look for potentially invalid object-nets constraints that emerge after deletion. These constraints must either be updated manually by an end-user or removed. In the end, all the changes must be persisted in a system.

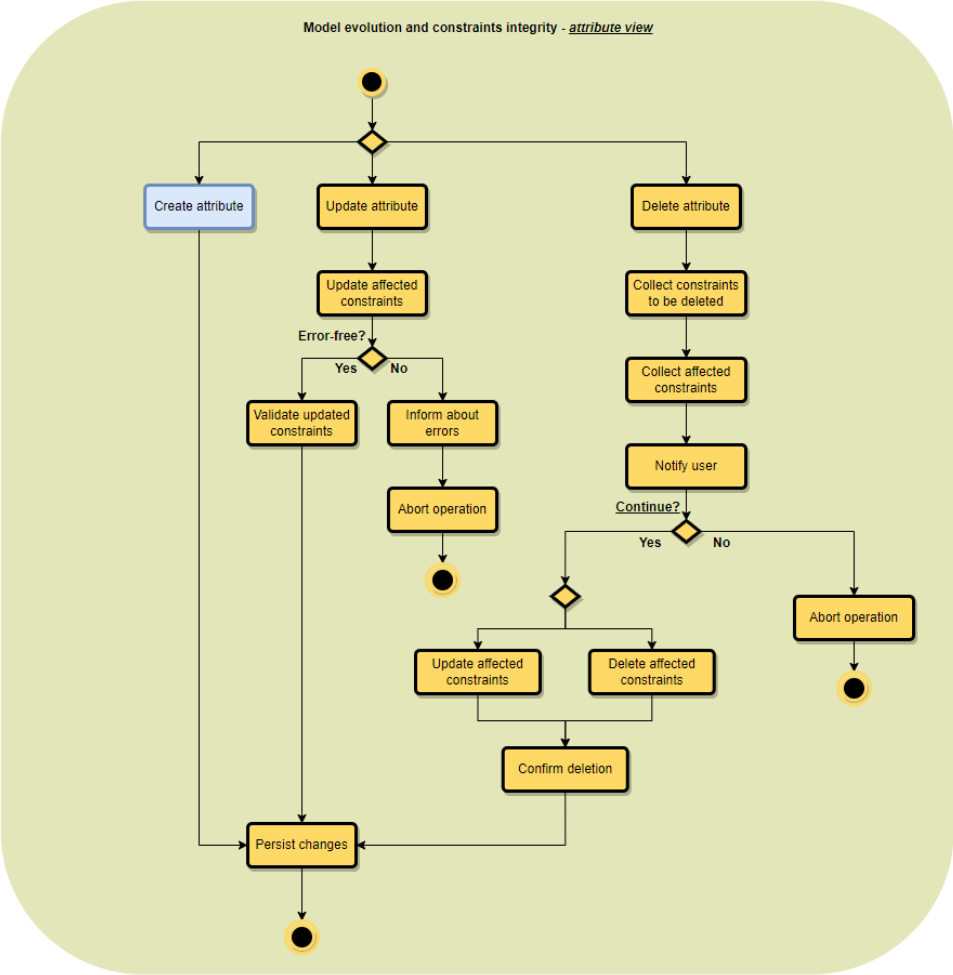

Figure 3.12: Attribute-related changes and constraints integrity.

Association changes and constraints at runtime

Any changes made to an association of a domain element demand a similar sequence of actions from a constraint engine. Creating a new association does not influence the validity of existing formulated requirements. Updating an existing association must trigger the automatic reevaluation of all constraints affected by the scope of an operation. If the engine is not able to perform an update, an end-user must be notified about the reason for an error. Deleting an association requires the manual user inference to manually correct all syntactically invalid object-nets constraints or delete them together with the association. In case of an error or a

user's intent, the deletion process could be aborted. Finally, upon successful removal, all the modifications must be persisted in a system. The detailed system in action for manipulating associations and delivering integrity mechanisms for constraints is depicted in figure 3.13.

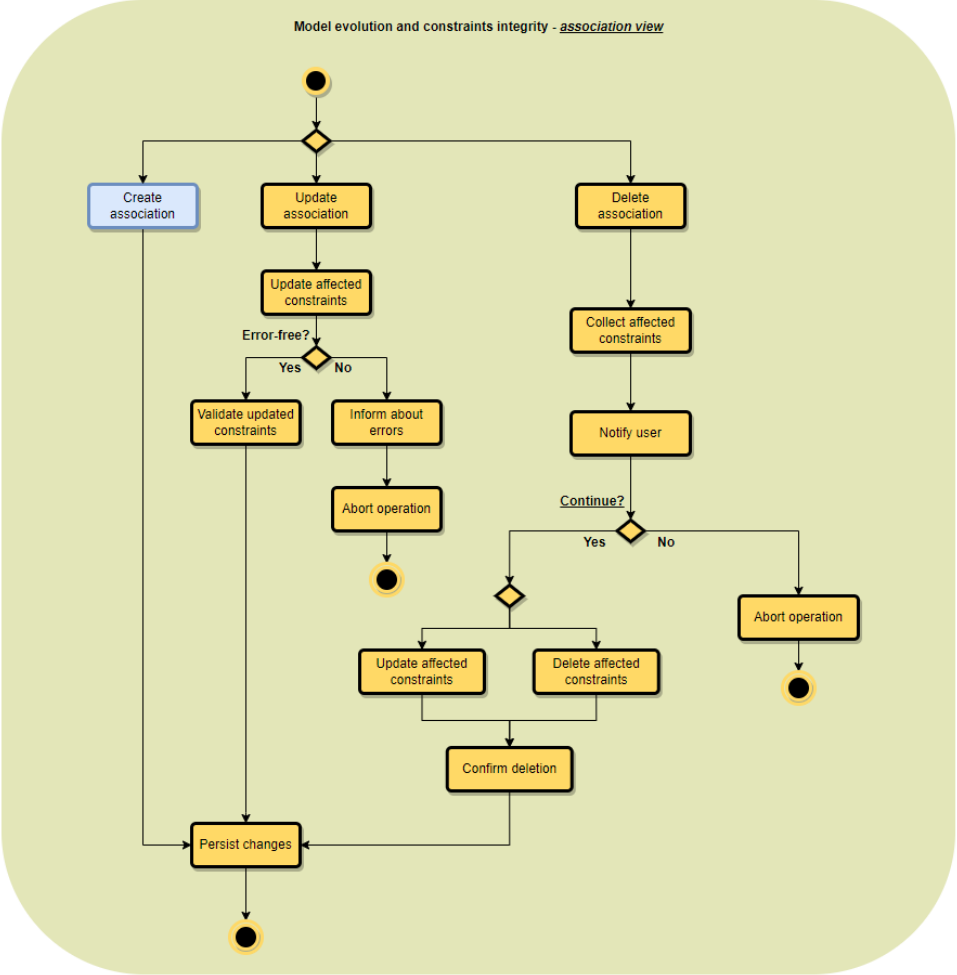

Figure 3.13: Associated-related changes and constraints integrity.

3.3.5 Constraints in the context of deep models

In a two-layered modeling concept, the constraints are specified on the classifier level and evaluated at the type of element instantiation. However, in some cases, it is complex or unnatural to present a domain element within such a flat hierarchy. To overcome this limitation, deep modeling blur the clear distinction between a type and an instance by introducing the notion of a clabject that allows designing entity hierarchies crossing multiple meta-levels [AK01]. A clabject possesses a dual facet and can be regarded simultaneously as a class element and instance. Figure 3.14 represents the domain sample of the project management system where the Software Engineer element is located on the bottom level of an ontological hierarchy and comprises all the traits of all its parent entities. In a world where a structural domain is refrained by additional requirements, constraining deep modeling entities is no exception. However, considering the problematic nature of infinite meta-levels in multilevel modeling, the following questions emerge for constraining such hierarchies:

1. At what point is an end-user allowed to define constraints?
2. At what level of the deep hierarchy should those constraints be evaluated?

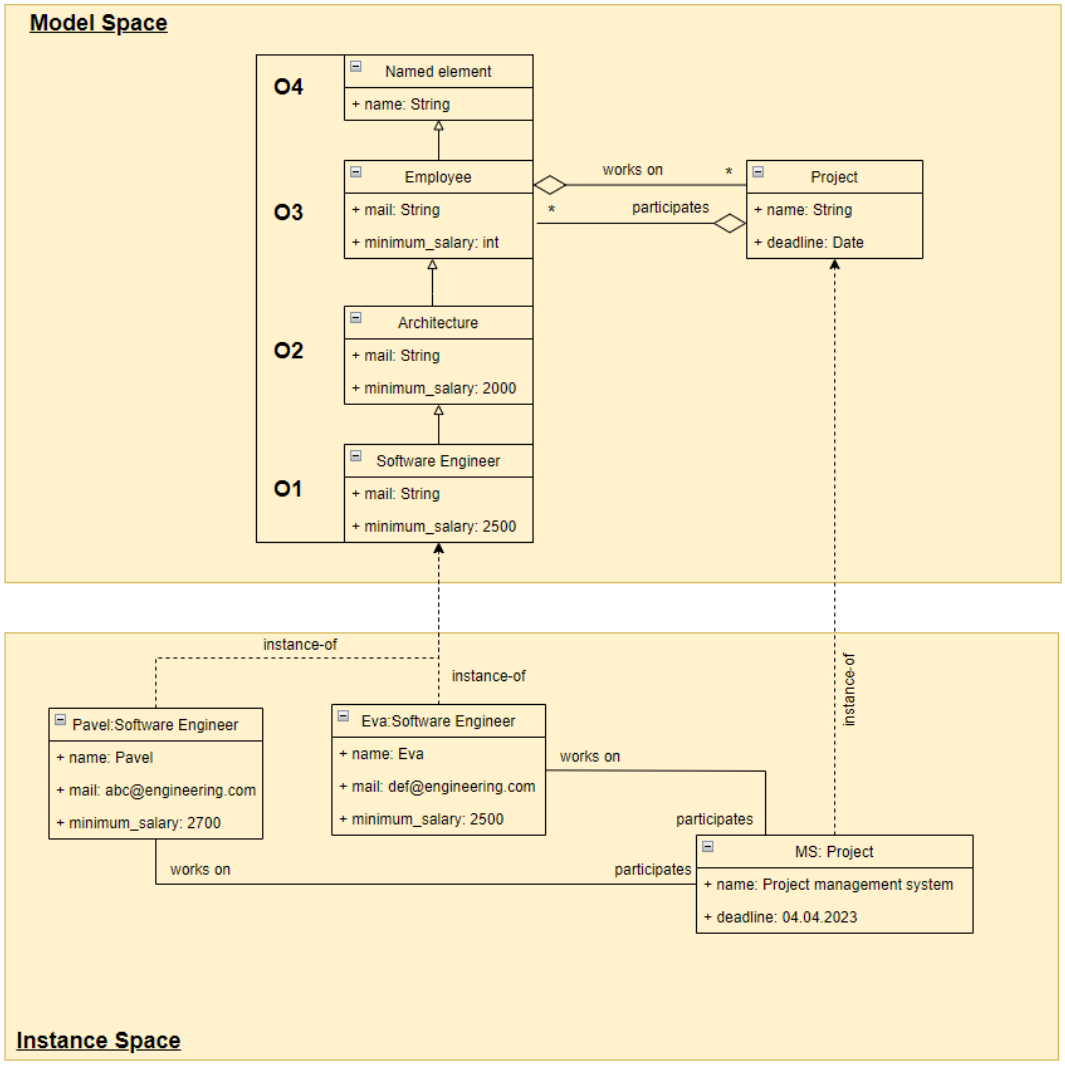

Figure 3.14: Deep modeling hierarchy - modeling and instance space.

3. Should a constraint engine be aware of a deep modeling hierarchy and be able to deal with them, or could it be flattened beforehand?

First, in order to handle the first question, the semantics of a clabject must be addressed one more time. According to the specification of OCL, the constraints can be applied to types of modeling elements [CG12]. Since every clabject has traits of a classifier, the system must be able to define constraints on any ontological level of a deep modeling hierarchy. Then, according to the deep instantiation mechanisms, all the constraints defined on the levels X, X+1, and X+n will be naturally inherited by all domain elements on the levels below. For instance, the *Software Engineer* class knows it is a deep instance of *Named Element*. Therefore, if some rules restrict the *Named Element* type, the *Software Engineer* entity must also be aware of those rules by inheriting them.

Secondly, the checking of constraint conformance happens at the time of instantiation. However, all clabject in a model space also have instance traits and can be populated with concrete values. An example of it is an instantiation of the *minimum salary* attribute on the O2 ontological level. If any constraint restricting this attribute exists in the deep modeling hierarchy, then according to the specification [CG12], this constraint must be checked already in the modeling space. On the other hand, creating a concrete instance of *Software Engineer* would also result in assessing the same constraint since it is inherited. Hence, to make the mechanism of

constraint evaluation consistent and avoid further complexities imposed on a modeling system, the decision is made to check the constraints only once at the type of object creation in instance space. It also implies that if the engine finds the violation of at least one constraint defined within a model space during the instantiation, then instantiation must be aborted with the respective notification delivered to an end-user.

Finally, it is vital to consider the boundaries of a constraint engine to be aware of a deep modeling hierarchy. Suppose the support for deep modeling is embedded in the constraint engine. In that case, it increases the learning curve of end-users who do not use multilevel modeling to design their domains. Moreover, the complexity of implementing and maintaining such functionality rises significantly. On the other hand, providing the "flattened" domain element creates an overhead for end-users that rely on deep modeling concepts because such frameworks would have to combine the final entity out of the whole deep hierarchy by themselves before delivering it to the constraint engine. To stay concept-agnostic and favor the flexibility of the future constraint engine, it is decided to discard the explicit support for deep modeling. As a result, the frameworks that use the concept of multilevel modeling must merge the entities from all ontological levels before delivering them to the constraint engine. The resulting flattened software engineer type available for applying constraints would look in the form shown in figure 3.15. It is worth noting that this is only a reference structure, and all the details are discarded here.

Flattened model element of "Software Engineer"

```
{
  "context": "Software Engineer",
  "attributes": [
    "name", // coming from Named Element
    "mail", // coming from Employee
    "minimum_salary" // coming from Employee
  ],
  "associations": [
    {
      "name": "works on", // coming from Employee
      "multiplicity": "*",
      "target": "Project"
    }
  ],
  "constraints": [
    {
      "name": "constraint_inherited_from_named_element",
      "definition": {...}
    },
    {
      "name": "constraint_inherited_from_employee",
      "definition": {...}
    },
    {
      "name": "constraint_one_defined_on_software_engineer",
      "definition": {...}
    },
    {
      "name": "constraint_two_defined_on_software_engineer",
      "definition": {...}
    }
  ]
}
```

Figure 3.15: Flattened deep hierarchy of Software Engineer

3.3.6 Technology independent API for domain constraints

Evaluating constraints is impossible without a reasoner component that evaluates them against concrete instances in a domain. The abundance of different query and constraint frameworks eliminates the need to implement the custom reasoner. However, every technology is other in its background concepts, rules, and concrete syntax. Therefore, the integration issue must be considered first before analyzing existing frameworks that tackle this problem in the following chapter. A straightforward approach is to find the best constraining or query-related framework at the current time and take it as a foundation of the logical level. However, the simplicity at first glance might not pay off in a more prolonged sense. What happens if the constraint framework reaches its end of support? What happens if the chosen technology cannot handle a new requirement that eventually appears? Should then the whole user-driven constraint component be rewritten from scratch? In order to avoid such issues in the future, it is necessary to write a technology-independent specification first.

The purpose of designing the specification of the constraint engine is to provide the means of standardization among any fundamental technology that will be chosen further for implementation. Moreover, it enables the abstraction of the system by omitting the implementation details and gluing to concrete technology. Therefore, by discarding the concrete-implementation-first approach, it is possible to switch over to the basic constraint-oriented technology easily. Finally, a well-defined specification allows the gradual enhancement of a system [Kru92]. Initially, a minor prototype might be implemented following the contract, but gradually it can be updated and extended, removing limitations and drawbacks.

Drawing an analogy with already existing specifications that are widely used as accepted standards, Java Persistence API (JPA) [KSK11] allows managing and persisting data between class entities and relational databases. JPA by itself does not define any concrete operations. It is just a contract that all object-relational-mapper (ORM) tools like Hibernate or Toplink follow while implementing it. If, at some moment, there is a need to switch the ORM library, it can be done with less effort.

The full definition of the specification for a user-driven constraint modeling framework will be proposed in one of the following chapters.

3.3.7 Formulating domain requirements via constraints by end-users

Finally, this section elaborates on the non-functional requirement of what means of defining constraints must be available in a user-driven constraint modeling system. The problem of constraint definition by non-experienced end-users is not novel. Many research studies focusing on the area of modeling like [TG07; Ber+10; YA16] claim that the constraint definition in a plain syntactic text format is unsuitable for users with no software background. The reason is the steep learning curve needed to obtain the necessary programming skills and familiarize yourself with the syntax.

A user-driven system may use a natural language for formulating domain restrictions to conceal the complexity of syntax. Considering this approach, a component with natural language support could synthesize a grammatically correct query from a simple text understandable for any human being [HFJ11]. However, as pointed out by [KB07], the real advantage of using natural languages comes with the close integration of it to a concrete domain. However, it is challenging to achieve in user-driven modeling since the domain will vary significantly from the context and necessary prerequisites.

As opposed to natural query languages, one could design an interactive graphical interface to formulate modeling constraints. Visual query systems offer means for graphical query specification. As summarized in the paper of [Rus+08], one of the following visual query systems may be considered instead of using the natural language technique:

- **Icon-based interface.** It offers a query construction by utilizing icons supplemented by meta query information to specify rules without referring to actual data. One of the systems that use such an interface is *VISUAL* ([Bal+96])
- **Form-based interface.** It possesses a set of structured components. Users only have to fill out predefined forms by the required criteria, and a query will be constructed automatically by a respective software vendor. One of the pioneers in this type of graphical query definition is *Query By Example* ([Zlo77])
- **Diagram-based interface.** It depicts a query construction using some geometrical figures presenting entities, often with connectors representing relations between them. *NITELIGHT* ([Rus+08]) can be taken as an example that enables query construction on web ontologies via diagrams.

To summarize, there is a need to design a graphical component that would assist an end-user in formulating domain constraints. However, since the user interface is not the primary focus of this thesis, implementing a visual component is deferred to future work.

3.3.8 Summary

This section summarizes all the requirements for the user-driven constraint modeling framework. Both functional and non-functional prerequisites for the final framework are described below.

Functional requirements

- **Comprehensive support for constraint types.** One of the key traits in expressing the requirements of a domain is the expressiveness of a constraint language. For this reason, the framework must support all the specified types defined in this section.
- **Support for extendability.** The framework must enable the definition of new constraint kinds at runtime to meet all trivial and non-trivial arising requirements from end-users.
- **Support for dynamic changes of a variant-aware model at runtime.** To prevent the age of constraints and their invalidation over time, the framework must define the means of dynamically adapting the correctness of rules.
- **Support for changing constraints at runtime.** The scope of restrictions specified for a domain model must be extendable on the demand of an end-user at runtime. It includes the dynamic creation, update, and deletion of domain restrictions.
- **Support for constraints validity and evaluation during instantiation.** The system must validate constraints during a model element instantiation. If an operation on the instance affects complex object-nets constraints, the system must analyze those constraints and either actualize them automatically or leverage them to an end-user.
- **Support for keeping constraints meaningful during the domain evolution.** Changes to the domain might affect the validity syntax of defined constraints. The solution must take it into account and offer ways to correct the limitation rules automatically or leverage it explicitly to an end-user.
- **Finite support for deep modeling.** The system must accept the flattened hierarchy of a multilevel modeling element. The constraints must be assessed during instantiation and comprise all rules defined in the whole deep domain element hierarchy.

Non-functional requirements

- Technology-independent API. The specification for a user-driven constraint modeling framework must be designed and implemented to encourage reusability and facilitate maintenance.
- Visual interface for constraint definition. In the long run, the system should provide a graphical interface to make the definition and composition of constraints effortless for end-users.
- Performance. Any operation related to constraints and their evaluation should happen within an acceptable time range without suspending the whole system or leaving it unresponsive.

List of Figures

Bibliography

[AGF13] Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. “Modeling language extension in the enterprise systems domain”. In: *2013 17th IEEE International Enterprise Distributed Object Computing Conference*. IEEE. 2013, pp. 49–58.

[AGK09] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. “A flexible infrastructure for multilevel language engineering”. In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 742–755.

[AGK12] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. “On-the-fly emendation of multilevel models”. In: *European Conference on Modelling Foundations and Applications*. Springer. 2012, pp. 194–209.

[AK01] Colin Atkinson and Thomas Kühne. “The essence of multilevel metamodeling”. In: *International Conference on the Unified Modeling Language*. Springer. 2001, pp. 19–33.

[AK03] Colin Atkinson and Thomas Kuhne. “Model-driven development: a metamodeling foundation”. In: *IEEE software* 20.5 (2003), pp. 36–41.

[Bal+96] Nevzat Hurkan Balkir et al. “VISUAL: A graphical icon-based query language”. In: *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE. 1996, pp. 524–533.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B France. “Models@ run. time”. In: *Computer* 42.10 (2009), pp. 22–27.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. “Model-driven software engineering in practice”. In: *Synthesis lectures on software engineering* 3.1 (2017), pp. 1–207.

[BD07] Achim D Brucker and Jürgen Doser. “Metamodel-based UML notations for domain-specific languages”. In: *4th International Workshop on Software Language Engineering (ATEM 2007)*. 2007.

[Ben+14] Nelly Bencomo et al. *Models@ run. time: foundations, applications, and roadmaps*. Vol. 8378. Springer, 2014.

[Ber+10] Gábor Bergmann et al. “Incremental evaluation of model queries over EMF models”. In: *International conference on model driven engineering languages and systems*. Springer. 2010, pp. 76–90.

[Béz04] Jean Bézivin. “In search of a basic principle for model driven engineering”. In: *Novatica Journal, Special Issue* 5.2 (2004), pp. 21–24.

[Béz05] Jean Bézivin. "Model driven engineering: An emerging technical space". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2005, pp. 36–64.

[BG01] Jean Bézivin and Olivier Gerbé. "Towards a precise definition of the OMG/MDA framework". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 273–280.

[BGS19] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@ run. time: a guided tour of the state of the art and research challenges". In: *Software & Systems Modeling* 18.5 (2019), pp. 3049–3082.

[Bor+20] Francis Bordeleau et al. "Towards model-driven digital twin engineering: Current opportunities and future challenges". In: *International Conference on Systems Modelling and Management*. Springer. 2020, pp. 43–54.

[Bro04] Alan W Brown. "Model driven architecture: Principles and practice". In: *Software and systems modeling* 3.4 (2004), pp. 314–327.

[CG12] Jordi Cabot and Martin Gogolla. "Object constraint language (OCL): a definitive guide". In: *International school on formal methods for the design of computer, communication and software systems*. Springer. 2012, pp. 58–90.

[CK90] Chen Chung Chang and H Jerome Keisler. *Model theory*. Elsevier, 1990.

[Cod07] Edgar F Codd. "Relational database: A practical foundation for productivity". In: *ACM Turing award lectures*. 2007, p. 1981.

[Cur+10] Carlo A Curino et al. "Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++". In: *Proceedings of the VLDB Endowment* 4.2 (2010), pp. 117–128.

[Da 15] Alberto Rodrigues Da Silva. "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155.

[Day90] Umeshwar Dayal. "Queries and views in an object-oriented data model". In: *Proceedings of the Second International Workshop on Database Programming Languages*. 1990, pp. 80–102.

[Fou+12] François Fouquet et al. "An eclipse modelling framework alternative to meet the models@ runtime requirements". In: *International conference on model driven engineering languages and systems*. Springer. 2012, pp. 87–101.

[FR07] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 37–54.

[FS17] Nicolas Ferry and Arnor Solberg. "Models@ runtime for continuous design and deployment". In: *Model-Driven Development and Operation of Multi-Cloud Applications*. Springer, Cham, 2017, pp. 81–94.

[GRE10] Iris Groher, Alexander Reder, and Alexander Egyed. "Incremental consistency checking of dynamic constraints". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2010, pp. 203–217.

[Hel+16] Rogardt Heldal et al. "Descriptive vs prescriptive models in industry". In: *Proceedings of the acm/ieee 19th international conference on model driven engineering languages and systems*. 2016, pp. 216–226.

[HFJ11] Lushan Han, Tim Finin, and Anupam Joshi. "GoRelations: An intuitive query system for DBpedia". In: *Joint International Semantic Technology Conference*. Springer. 2011, pp. 334–341.

[HT06] Brent Hailpern and Peri Tarr. “Model-driven development: The good, the bad, and the ugly”. In: *IBM systems journal* 45.3 (2006), pp. 451–461.

[JB06] Frédéric Jouault and Jean Bézivin. “KM3: a DSL for Metamodel Specification”. In: *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer. 2006, pp. 171–185.

[KB07] Esther Kaufmann and Abraham Bernstein. “How useful are natural language interfaces to the semantic web for casual end-users?” In: *The Semantic Web*. Springer, 2007, pp. 281–294.

[KRS15] Filip Křikava, Romain Rouvoy, and Lionel Seinturier. “Infrastructure as runtime models: Towards model-driven resource management”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2015, pp. 100–105.

[Kru92] Charles W Krueger. “Software reuse”. In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[KSK11] Mike Keith, Merrick Schincariol, and Jeremy Keith. *Pro JPA 2: Mastering the JavaTM Persistence API*. Apress, 2011.

[KW07] Stephan Kurpjuweit and Robert Winter. “based Meta Model Engineering.” In: *EMISA*. Vol. 143. 2007, p. 2007.

[May+17] Heinrich C Mayr et al. “Model centered architecture”. In: *Conceptual modeling perspectives*. Springer, 2017, pp. 85–104.

[OA15] Janis Osis and Erika Asnina. “Is Modeling a Treatment for the Weakness of Software Engineering?” In: *Handbook of Research on Innovations in Systems and Software Engineering*. IGI Global, 2015, pp. 411–427.

[Par+20] Paolo Pareti et al. “SHACL satisfiability and containment”. In: *International Semantic Web Conference*. Springer. 2020, pp. 474–493.

[PLS14] Eric Prud’hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. “Shape expressions: an RDF validation and transformation language”. In: *Proceedings of the 10th International Conference on Semantic Systems*. 2014, pp. 32–40.

[Rei81] Phyllis Reisner. “Human factors studies of database query languages: A survey and assessment”. In: *ACM Computing Surveys (CSUR)* 13.1 (1981), pp. 13–31.

[Rum05] James Rumbaugh. *The unified modeling language reference manual*. Pearson Education India, 2005.

[Rus+08] Alistair Russell et al. “Nitelight: A graphical tool for semantic query construction”. In: (2008).

[Sel07] Bran Selic. “A systematic approach to domain-specific language design using UML”. In: *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07)*. IEEE. 2007, pp. 2–9.

[Sol+00] Richard Soley et al. “Model driven architecture”. In: *OMG white paper* 308.308 (2000), p. 5.

[SS09] Renuka Sindhgatta and Bikram Sengupta. “An extensible framework for tracing model evolution in SOA solution design”. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, pp. 647–658.

[SZ16] Michael Szvetits and Uwe Zdun. “Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime”. In: *Software & Systems Modeling* 15.1 (2016), pp. 31–69.

[TG07] John TE Timm and Gerald C Gannod. “Specifying semantic web service compositions using UML and OCL”. In: *IEEE International Conference on Web Services (ICWS 2007)*. IEEE. 2007, pp. 521–528.

[Vie+21] Michael Vierhauser et al. “Towards a model-integrated runtime monitoring infrastructure for cyber-physical systems”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2021, pp. 96–100.

[WK03] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[YA16] Tao Yue and Shaukat Ali. “Empirically evaluating OCL and Java for specifying constraints on UML models”. In: *Software & Systems Modeling* 15.3 (2016), pp. 757–781.

[Zlo77] Moshe M. Zloof. “Query-by-example: A data base language”. In: *IBM systems Journal* 16.4 (1977), pp. 324–343.

[ZX16] Yunxiang Zheng and Youru Xie. “Metamodel for evaluating the performance of ICT in education”. In: *International Conference on Blended Learning*. Springer. 2016, pp. 207–218.