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

04.04.2023

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, 04.04.2023

Anton Skripin

Abstract

In today's fast-paced technological landscape, businesses and organizations require flexible and adaptable software systems that are able to respond dynamically to highly adaptive user needs and requirements. The integral part of any modern software is a domain layer that encapsulates the business rules and data of a target system into a set of entities and interrelations among them. However, mere models are often insufficient to catch all types of enterprise requirements, especially if they emerge in real-time and systems must adjust dynamically while still remaining available. This thesis presents the concept of user-driven constraint modeling that enables end-users to enrich dynamic adaptive models with constraints at runtime. It considers different groups of potential users with distinct levels of expertise. The highlight of this work is a framework that permits the intuitive and effective formulation of constraints on a domain layer at runtime and validates them at runtime. In addition, the dynamic nature of adaptive models is also taken into account. Therefore, the constraint space is kept up-to-date by employing the following techniques like constraint synchronization and backward links. The framework's design is modular and extensible, allowing easy integration with existing software systems. The primary goal of this thesis is to prove that end-users should be an integral part of the software evolution process by tailoring the domain layer dynamically via constraint definition and validation.

Contents

1 Introduction

1.1 Motivation

The prevalence of software in our daily routines is undeniable. The scope of potential usage ranges from the applications that we use on our smartphones to the complex critical systems that drive business activities and governments. Since the complexity of software products increases, it creates certain challenges when it comes to designing, developing, and maintaining them. On the other hand, integrating models into the software lifecycle helps to tackle the intricacy of complex software by abstracting and simplifying parts of a system. As stated by [Hel+16], there are two general types of models: descriptive and prescriptive. Descriptive models aim to describe the characteristics and behavior of a system. Prescriptive models are designed to specify how an entity should be constructed. Both types of models are crucial because they provide ways to define and enforce standards as well as to describe the actual behavior of an object under study.

Modern software is not just about lines of code. On the contrary, it is a system consisting of numerous interconnected artifacts. Thus, requirements are collected from involved stakeholders and reflected in design and system architecture. Design decisions and architecture shape the actual implementation. Finally, generated or created documentation maintains the origin of software knowledge among involved parties. All of these artifacts are necessary to develop and support a system during its software lifecycle. A software program cannot be considered complete if it is missing any of the previously mentioned components; otherwise, it would merely function as some purpose-specific script.

Nevertheless, what overarching role do models play during the development of software? First, a model is a link between a client and a developer serving as an intermediate component that helps to understand a system. Additionally, the use of models in conjunction with documentation plays a vital role in maintaining the identity of software 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, Unified Modeling Language (UML) [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 structural

and behavioral models produced by MDA techniques reflect the entire development cycle of a system. As mentioned by [FR07], these models form a layer of abstraction above the code level. Unfortunately, one of the limitations of MDA hides in its specification. It assumes that the system requirements are fully gathered and determined upfront. Unfortunately, it is often not the case in practice because new requirements tend to appear as a system evolves. This leads to difficulties while adapting a system to changing requirements or integrating new functionality on the fly. To address these restrictions, models can be used at runtime to adjust software systems in highly variable environments. By casually connecting a model with a running system, the state of software and its behavior can be captured and influenced to a certain extent by adapting the model instead of code. [BBF09]

Despite a rich scope of application and increasing potential for use in the future, mere models are not enough expressive to catch all functional and non-functional requirements. One of the ways to circumvent this issue is to enrich a model by applying constraints. Up until nowadays, the research community encourages the usage of constraints as a supplement to ensure that a model accurately represents desired specifications at design and development time. In a similar fashion, applying constraints on models at runtime would allow adapting a system in a controlled way, ensuring that software keeps operating within safe and valid bounds. If we imagine a system where one can constrain a dynamic model at runtime, the following questions arise. Does it make sense to shift the process of defining and updating constraints to end users? If the answer is yes, how must such a system look like? By what means can those constraints be formulated by users, and how should they stay consistent in the presence of dynamic changes? This thesis attempts to reveal answers to these topical issues.

1.2 Problem Statement

As a model-centric technique, MDA has proven to be effective in developing high-quality software with improved cost-effectiveness. However, this approach does not address the problem of software evolution. Indeed, to integrate a new component into 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 system together, breaking down the barrier between software development and deployment becomes possible. A promising solution to such a matter is to extend the spectrum of a model from static to runtime and perceive models as running software artifacts. [FS17; May+17]

When mentioning the role of models during runtime, it is essential to set clear boundaries and distinctions between interpreted models, models at runtime, and adaptive entity models. As such, interpreted models are analyzed by an interpreter rather than being compiled directly into machine code. The advantage over compiled models lies in their flexibility and adaptation without having to recompile the entire program [Fum+17]. Models at runtime are models that are used during the operation of a system. Runtime models must fulfill the definition of self-adaptation and self-reflection by monitoring the state of software and changing it according to some rules [FS17]. Finally, adaptive entity/application models [Keg27] refer to models that are designed to adapt to changing environmental data over time. The concept of adaptive entity models stems from the "Adaptive Object Model" (AOM) pattern [YJ02], where the domain components can be created dynamically while a system is running by lifting the level of abstraction and allowing users to interact with the meta-model of classes and instances. Using adaptive models in the domain layer of software results in a user-centric approach, as such systems can be guided by external modifications in model elements and entities, respectively. Hence, the primary focus of this work is to research and explore user-driven techniques on how adaptive entity models should be constrained at runtime to ensure the conformance of requirements expressed by end-users.

To demonstrate the benefit of dynamic runtime models over traditional software development techniques, we depict the same process of creating a software product but with distinguishable required efforts. Figure 1.1 showcases the lifecycle of software utilizing MDA mechanisms, while figure 1.2 reveals a more advanced workflow with the support of adaptive models.

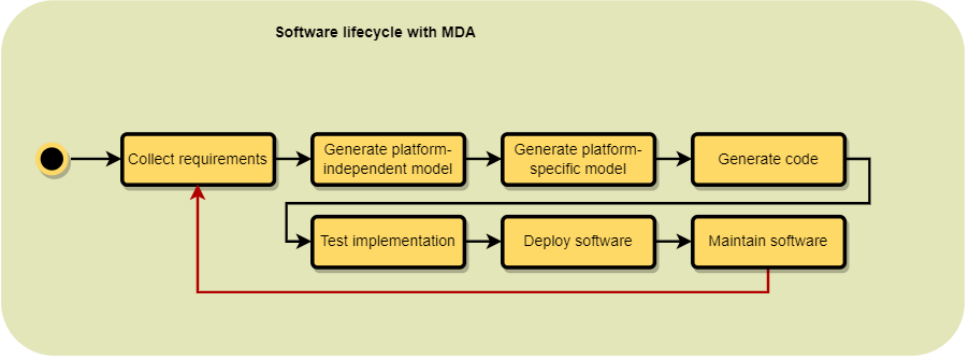


Figure 1.1: Software development lifecycle with MDE

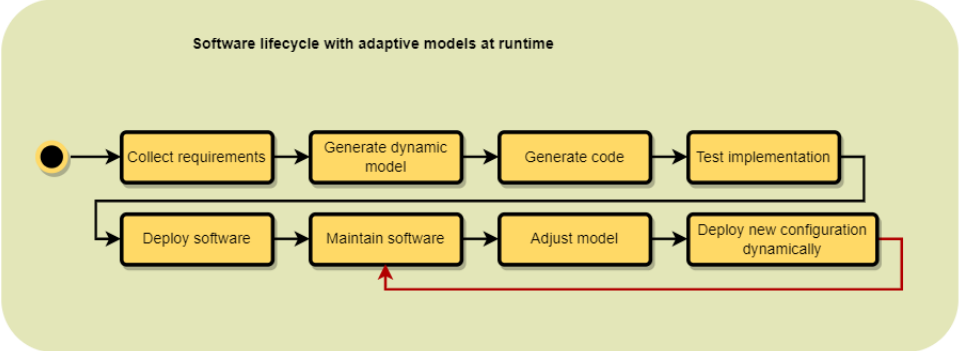


Figure 1.2: Software development lifecycle with dynamic models

Regardless of the chosen technique, the creation of software starts with gathering and analyzing requirements. The next step is modeling. In the context of MDA, modeling requires constructing a platform-independent model (PIM) followed by generating a platform-specific model (PSM) that represents a system in the context of a specifically implemented technology. If we focus on dynamic models, modeling involves generating runtime models that reflect the domain of a system at runtime. Such models are dynamic by their nature and allow evolving the domain layer of a system at runtime. As the next steps, both approaches require the code either to be generated or implemented manually, testing the system, and, finally, its deployment.

Moreover, both figures share the same recurring block representing software maintenance. The crucial advantage of dynamic models is that the system can be reconfigured dynamically to meet new requirements without having to take the system offline. This enables greater flexibility and agility in the software development process, allowing the system to evolve over time in response to changing requirements and user needs. In contrast, the MDA approach requires software redeployment that could poorly affect business requirements. Moreover, as seen in Figure 1.2, in comparison to Figure 1.1, the cost of implementing new functionality is reduced. As such, minor changes that trigger automatic model reconfiguration lessen the burden on software engineers, thus allowing them to focus on implementing more complicated infrastructure.

Despite showing significant advantages against traditional model-driven software development, dynamic models at runtime should also be created in such a way that it would be possible

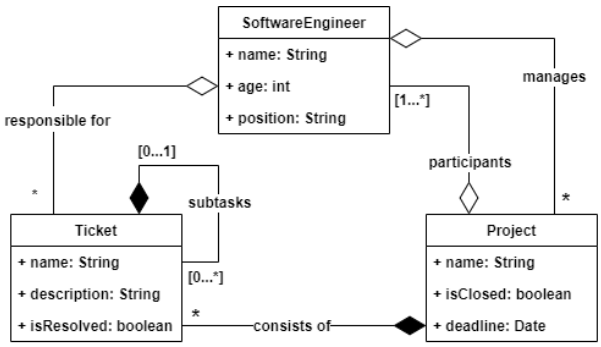


Figure 1.3: Trivial management system class diagramm

to meet functional and non-functional requirements. To elaborate the problem space further, figure 1.3 shows a simple project management system model that reflects the domain of 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. However, the following constraint written in OCL helps us eliminate this drawback.

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

As a result, the necessity to control the evolution of a domain model within allowed and realistic bounds by applying constraints leads to three major problems: language constraint expressiveness, constraint definition, and constraint-space integrity. Each of the stated issues is described below.

First, in order to be able to enrich the notion of adaptive models at runtime, it must be clear what types of constraints are required and desirable. In other words, the evaluation and analysis should be performed in order to select the set of constraints that make sense to be used at runtime by end-users. The list of constraints should be compact but sufficient to formulate various types of requirements ranging from trivial attribute-based to complex association constraints.

Secondly, different groups of interested stakeholders might need to formulate restricting rules on a domain model. It is clear that people with no programming expertise might struggle to constrain the domain in full using declarative tools sticking 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 involved parties dealing with the raw syntax of specification.

Finally, another issue stems from the dynamic nature of runtime models. If at some moment in time *T*, there is a new requirement to encapsulate the state of a project into a separate entity, 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. Hence, changes that a domain model undergoes at runtime might bring inconsistency to the space of constraints if synchronization mechanisms are absent.

1.3 Objective

This thesis attempts to fill the research gap in the area of runtime models that are used in the domain layer to dynamically adapt entity structure. Therefore, this thesis aims to research and evaluate the viability and possibility of utilizing user-driven techniques to constrain models

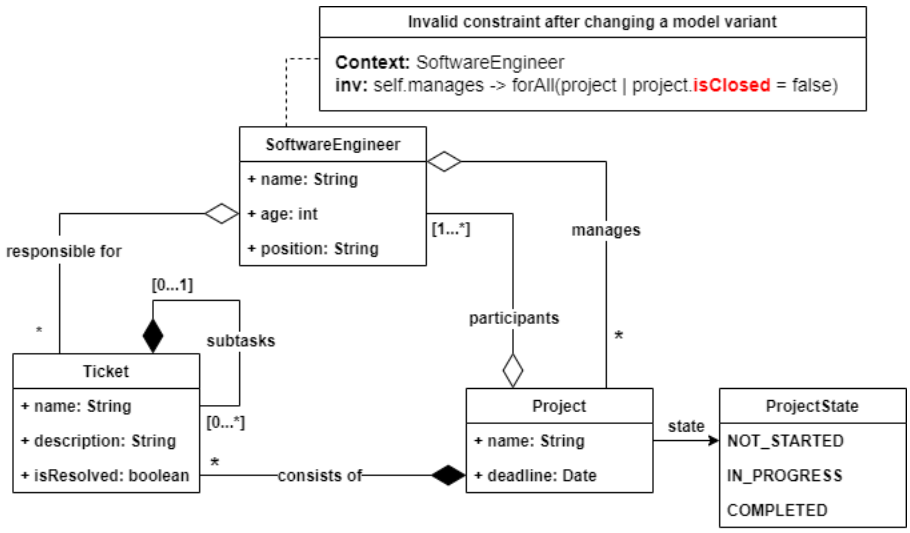


Figure 1.4: Updated model variant results in invalid constraint

at runtime. Subsequently, this work summarizes state-of-the-art regarding various constraint and query engines as well as modeling techniques that are essential for addressing the challenges outlined in the previous section. Finally, this thesis introduces a framework that allows end-users to impose constraints on a dynamic model at runtime and provides synchronization mechanisms to keep constraints up-to-date according to the present domain specification.

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

1. Introduced the concept of user-driven constraint modeling for models at runtime. This concept is elaborated and implemented as a specification for restricting adaptive models by users with no preliminary background in software engineering.

2. Proposed a mechanism to enrich a domain model with constraints by end-users. The offered strategy takes into account the dynamic nature of runtime models. Hence, additional efforts are made to keep the constraints valid in the presence of the dynamic evolution of domain entities.

3. Implemented a framework primarily focusing on the novel concept of end-user-driven constraints at runtime. While various tools permit formulating constraints intuitively, none of them, to the best of our knowledge, allows shifting the constraint definition and validation process from design to runtime, thus making it part of the domain layer itself.

Finally, the following research questions are addressed in the scope of this work:

1. Would it be reasonable to delegate the responsibility from software developers to domain experts and other end-users to formulate constraints on an entity model at runtime?

2. What mechanisms and techniques would be required to keep constraints up-to-date in the presence of domain changes at runtime?

3. How could a user-driven constraint modeling system be integrated into existing software tools and workflows to enhance the overall software development process?

4. How could the performance of the user-driven constraint modeling system be optimized to ensure it is scalable and responsive even if the number of model elements is significant?

5. Would it be possible to have such a system loosely coupled and encapsulate the implementation into a standalone module so that it could be easily embeddable into other software?

2 Foundation

The field of model-based software engineering has gained significant attention in recent years as a promising technique to improve the quality, efficiency, and reliability of software products. This chapter gives an overview of key concepts and techniques that build a solid foundation required for further implementing a user-driven constraint modeling framework. At first, we focus on models and metamodels. Then we discuss how modeling techniques help to design and create modern software in an efficient and cost-effective manner. Moreover, we dive into more specialized topics like multilevel models and runtime models. Finally, various constraints and query languages are overviewed that are essential for verifying the properties of a model.

2.1 Models and metamodels

Modeling is a concept that is broadly used in various fields. Many representatives of different areas of occupations, like sociologists, physicists, chemics, mathematicians, and others, construct models in order to abstract some concepts and, by doing so, facilitate the understanding of a target object. Without any doubt, the increasing software complexity and the need to exchange information between domain experts and developers are key factors that make models 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 categories of utilization according to their purpose. Although the domain of a model varies depending on the field of application, in general, it is possible to identify three common groups of models. The first group helps to grasp some domain knowledge and documentation of software artifacts. The next one helps in examining a system under study and depicting its characteristics. 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 definition 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, Petri Nets, 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 purpose and is domain-oriented, its power lies in its ability to be concise while still expressive.

Nevertheless, the general 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 decides 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 in the context of a particular domain.

On the other hand, a functional distinction between DSLs and GPLs is noticeable. A generalized language imposes some limitations on an engineer to define a domain. The reason is that since GPLs are designed to be flexible and adaptable to various contexts and applications, they may lack some necessary concepts to comprehensively define a particular domain. 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 concerns [Sel07]:

1. refining a language to a particular domain requires a lot of financial, time, and human resources

2. lack of adequate tool support required for developing software, while general purpose languages typically have the support of a compiler, debugger, and transformation tools required for integration

3. lack of approval and integration of third-party libraries, unlike general-purpose languages

Because of the obstacles mentioned above, some GPLs provide means for customization and extension of a language to be closer to a domain. As an attempt to eliminate the need to create custom DSLs, UML provides generic extension mechanisms that can be used to tailor a domain with field-relevant concepts. 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 have advantages and disadvantages, and the final choice of a tool might depend on various business-specific factors.

2.1.2 Essense of metamodeling in software development

Complex software systems nowadays tend to be characterized by multiple models residing on different conception 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 that can be used by instances of such meta-elements. In other words, a metamodel is a model of a model [KW07]. As such, the whole notation of UML and its description is 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 model elements 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 a meta-hierarchy.

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

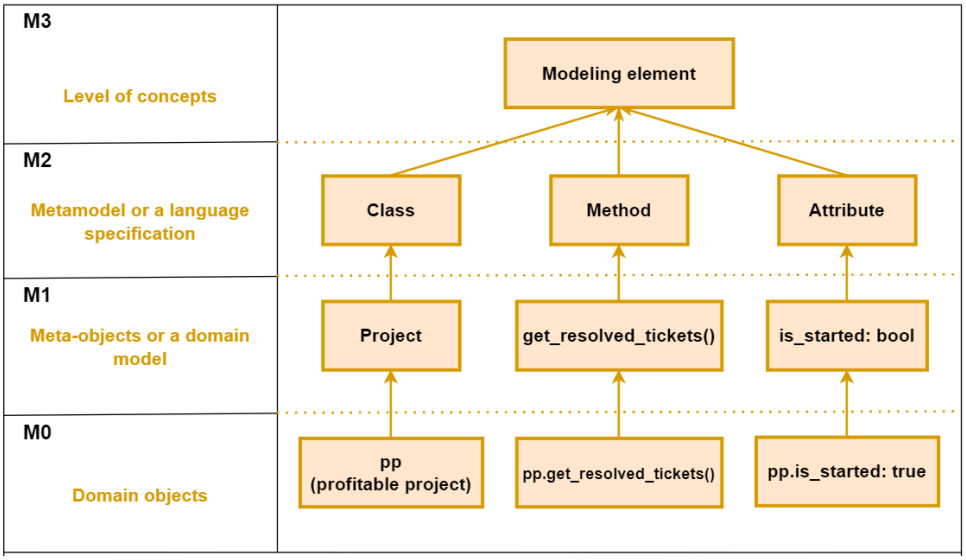


Figure 2.1: Classical meta-hierarchy with digital twins

- 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
- 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

In conclusion, it is essential to understand the relation link between a model and a meta-model. 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 semantically 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 concepts to be instantiated on a level below [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 can always find its definition in its metamodel.

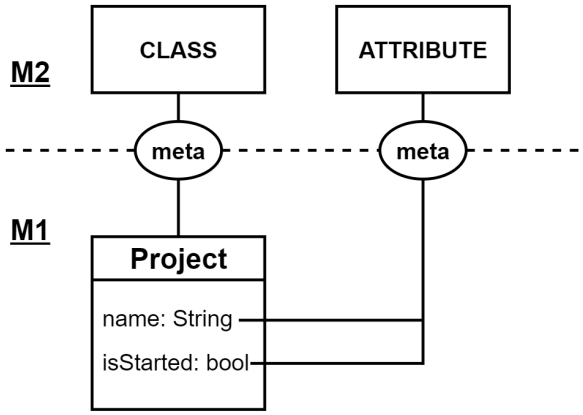


Figure 2.2: Metamodel to model relation

2.2 Model-driven Engineering

Creating modern software involves several stages that continuous and must be linked to each other. Those stages represent the software lifecycle. It starts from gathering requirements and completes by 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 steady 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 the system where the code base is not backed up by architecture, domain, and requirement artifacts? 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 using round-trip engineering.

A promising solution to overcome the modern challenges of developing industrial software systems is Model Driven Engineering (MDE). The continuous increase in software complexity, the ongoing demand for higher product's 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 and derivation of new models from metamodels [AK03].

As claimed by [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 consists of 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 encourages 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 facilitate the reuse of artifacts. Secondly, generated and non-generated artifacts are also essential parts 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 mentioned above, models play a central part in creat-

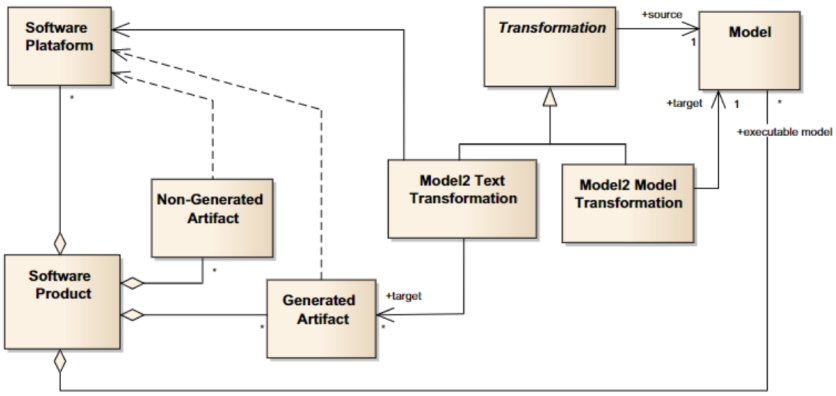


Figure 2.3: Software product composition with MDE, taken from [Da 15]

ing software in MDE. Models can be manually created by domain experts, designers, 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 approach. The overall process of software development with MDA 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 [Gro06] or ATL [Jou+08] perform the transition from more to fewer abstraction views of software 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) [Gro18] - a meta-meta language for all meta-models [Béz04].

However, even though MDA is regarded as a paradigm switch from the long-established idea 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 driven by requirements. Unfortunately, OMG has no precise formalization about the transformation of gathered requirements from a computation-independent to a platform-independent

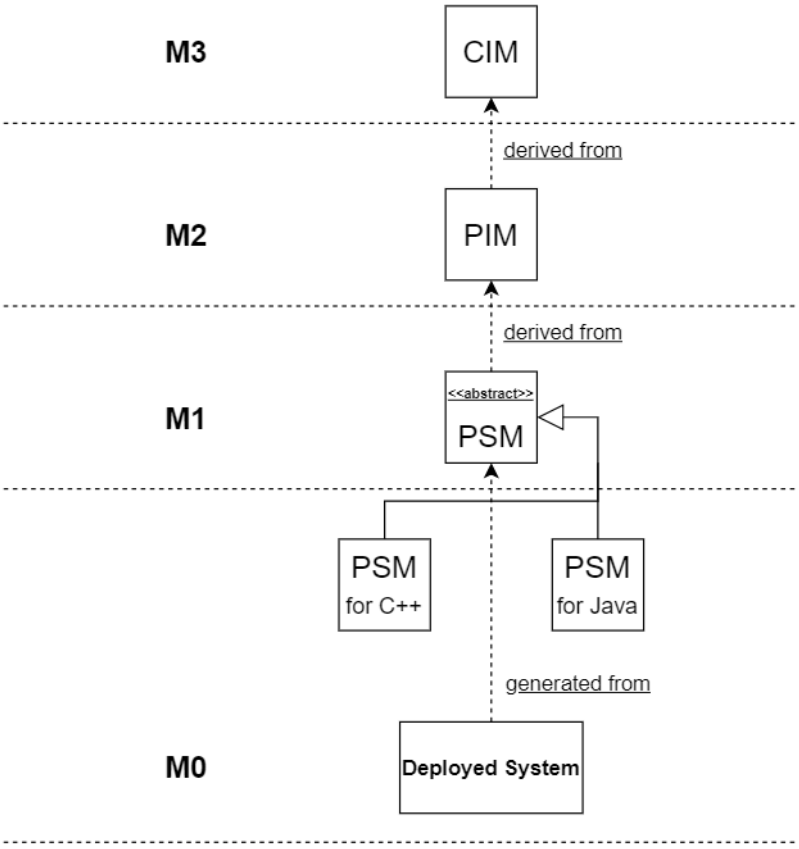


Figure 2.4: Artifact generation with MDA

view. Another author [HT06] argues that the MDA imposes an abundance of multiple distinct abstraction views on a system that demands 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 indeed reduces complexity or just 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

The demand for modeling tools to conform to rapidly changing requirements and production of concise yet comprehensive domain languages is rising. This leads to an increasing demand for improved capabilities to express domain needs by MDE modeling tools that rely on MOF as their primary meta-metamodel. However, the traditional metamodel hierarchy that is at the core of modeling artifacts might create obstacles due to its restricted support for extension mechanisms. As summarized by [AGF13] and elaborated further below, the standard extension mechanisms might solve the problem, but all of them tend to increase the overall complexity of a final 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 let customize a target model closer to a target domain.

However, it implies limitations on compatibility if every modeling language defines its own 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 enriching a model. 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 factory. This class hierarchy is present in figure 2.5. One can observe that the elements are both instances of some language definitions (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 to provide 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.

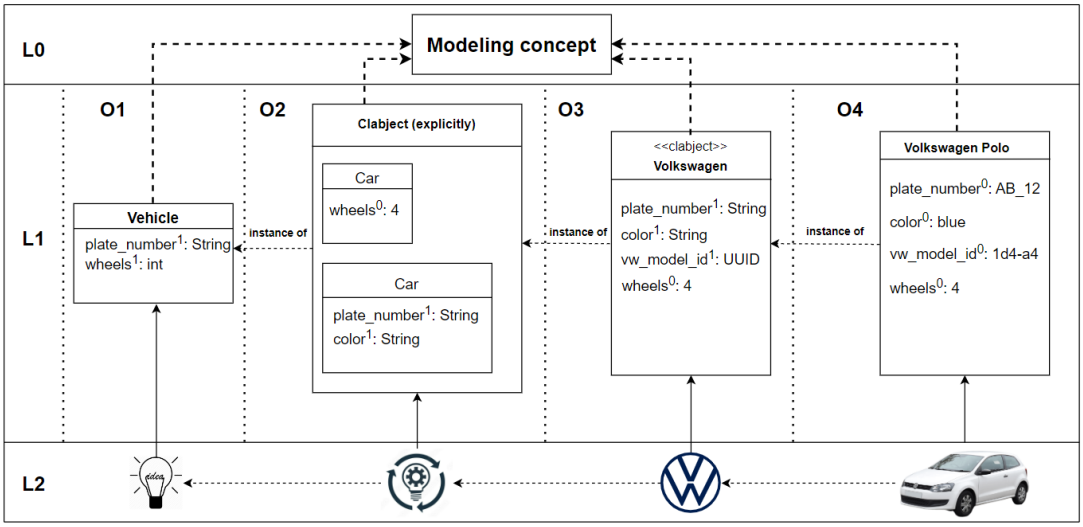


Figure 2.5: Deep model of a vehicle manufacturing plant

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, one of the following chapters elaborates on a more detailed classification of constraint languages and their importance in entity model hierarchy.

2.4 Models at runtime

The emergence of new types of software that operate in highly dynamic environments requires such 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 state adaptation 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 shifts the paradigm of creating software from code-centric to model-centric by raising the level of abstraction while implementing systems. 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. Such models that represent the reflection level of a running system are known in the research community as models@runtime [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 mandates particular prerequisites

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 in terms of consuming resources a runtime engine is. Running it on small devices might be impossible if the memory footprint is considerable. 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 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 a greater number of concrete disciplines where runtime models could be applied.

System architecture with models at runtime

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

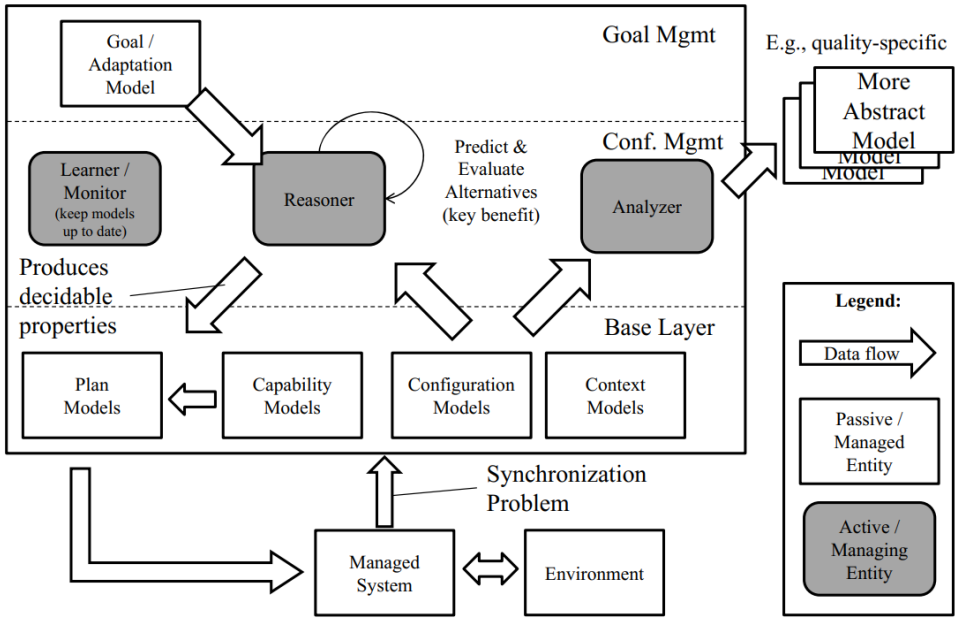


Figure 2.6: Generic architecture for models@run.time systems, taken from [Ben+14]

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

With the increasing demand to design and implement software components generically and flexibly adhering to changing requirements, the research and industrial techniques to achieve

software variability are becoming more critical. As a result, in an ever-changing business environment, software systems must adapt to varying market necessities, user preferences, and other factors. Therefore, it requires some software strategies that can efficiently produce variants of the software system, each tailored to particular needs.

One of the options to address such issues is to use so-called adaptive-object models (AOM) [RTJ00]. AOM is an architectural pattern in which classes, attributes, and associations are presented as meta elements. A system that employs such a pattern interacts only with the meta-elements. As a result of applying this technique, users are able to model parts of the domain dynamically at runtime.

Compared to a classical approach for creating software using the object-oriented paradigm, adaptive object models differ in how parts of a system are constructed and adjusted. In object-oriented design, classes shape the structure of the whole system. Business requirements are reflected in a class model. Hence, implementing new functionality results in code adjustments and a new version of an application. On the other hand, AOM describes domain entities not as static classes but instead as instances of meta elements, thus allowing them to be executed and extended at runtime. As a result, it enables quick and efficient adjustments reflected in a running artifact whenever changes are required [YJ02].

As noted by [YJ02], the UDP framework can be an example of applying AOM architecture in practice. By using the principles of AOM, this framework makes it possible for users to create new complex object entities from existing components at runtime. Such dynamic composition is achieved by utilizing attributed composite objects to specify and manipulate entities at runtime. The primary architecture of the UDP framework is shown in figure 2.7. By instantiating *ComponentType*, new domain entities can be created dynamically on demand. The instantiation of a new type leads to the definition of attributes through *AttributeType*. A strategy pattern helps define a particular entity's dynamic constraints and behavioral rules. In addition, to allow complex contract definition by composing several rules, they can be nested using the composite pattern.

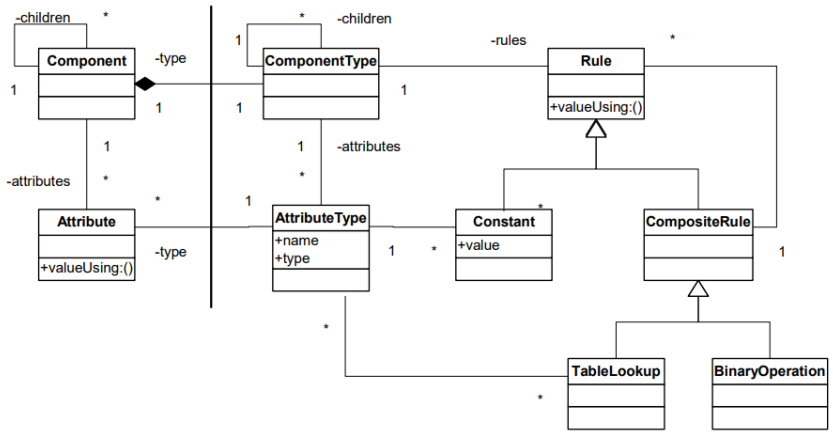


Figure 2.7: UDP framework architecture integrating AOM, taken from [YJ02]

Despite providing an opportunity to develop software parts at runtime, AOM has some drawbacks. One of them is related to the evolution of a system and data consistency management. As software products tend to evolve by adapting to emerging needs, the migration of data and code is necessary. If changes concern the modification of the domain level of software, the delivery of a new variant is required. The research community has suggested various methods to offer migration techniques that rely on end-users when fully automated migration is impossible [Hen+10; FCA09; MK15]. However, as mentioned by [Keg27], employing such approaches might lead to model inconsistency if significant portions of the model are adjusted and refactored. The authors [Keg27] propose a novel mechanism, based on the AOM con-

vention, to prevent certain parts of a domain from becoming outdated during runtime. This mechanism enriches the AOM pattern by endowing every instantiated model element with the ability to retain the structure of a model element used at the time of instantiation. As a result, the proposed mechanism ensures that the underlying domain stays integrity consistent even during major dynamic updates.

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 2.8 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, in reality, this relationship might be restricted 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 requirements, it should be defined via textual constraints.

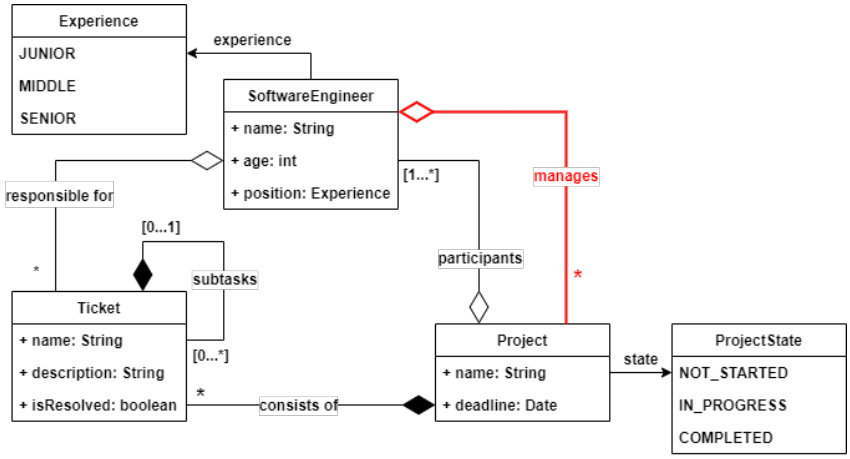


Figure 2.8: Project management system domain model

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 constrain 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. Hence, the following constraint categories could be classified [CG12]:

- invariant restrictions on a class level that must be satisfied with every instantiation from a constrained classifier
- rules which every derived field should satisfy 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 2.9: OCL 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

```
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 2.10: SPARQL constraint for the project management system

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 2.11 below.

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

```
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 2.11: Algorithm for model change adaptation at runtime, taken from [GRE10]

elements: deep models, an emendation service, and an impact analyzer.

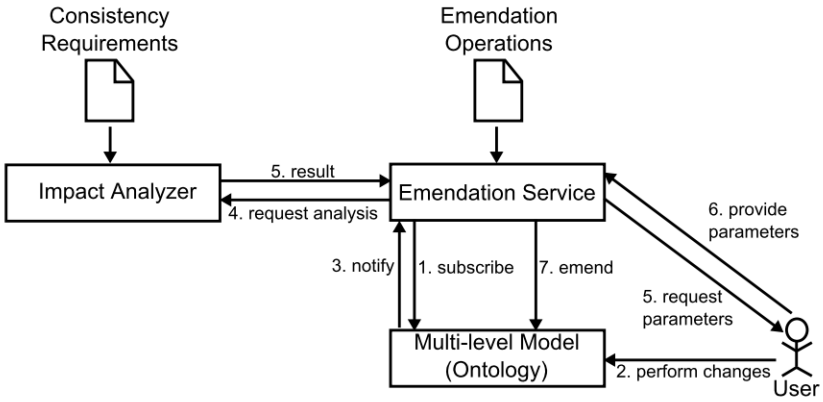


Figure 2.12: Emendation service architecture, taken from [AGK12]

As shown in figure 2.12, 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 reference link to an end-user for external feedback and guidance.

3 Concept and requirements

In this chapter, we explore the primary requirements for a system that should support constraint definition, evaluation, and modeling in the presence of runtime changes and entity-relational model variability. At first, we analyze the various types of constraints that end-users may require while constraining a domain model. Subsequently, we describe fundamental workflows for constraint definition and modeling. Finally, we summarize the key aspects of a future system that will serve as the foundation for the future design and implementation chapters.

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 on a domain model. For this reason, the domain model of a project management system mentioned in the previous chapters is reintroduced. This domain of the management system will be used as a study model for further analysis. However, the initial simplicity of the 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 status options are available. The difference between them is that the *premium status* permits demanding additional requirements while requesting a product compared to the restricted *basic status*.
- ***Project*** - an entity of a managed project in the real world. Every project has a unique name and deadline. It consists of numerous sprints and is developed by software engineers.
- ***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.

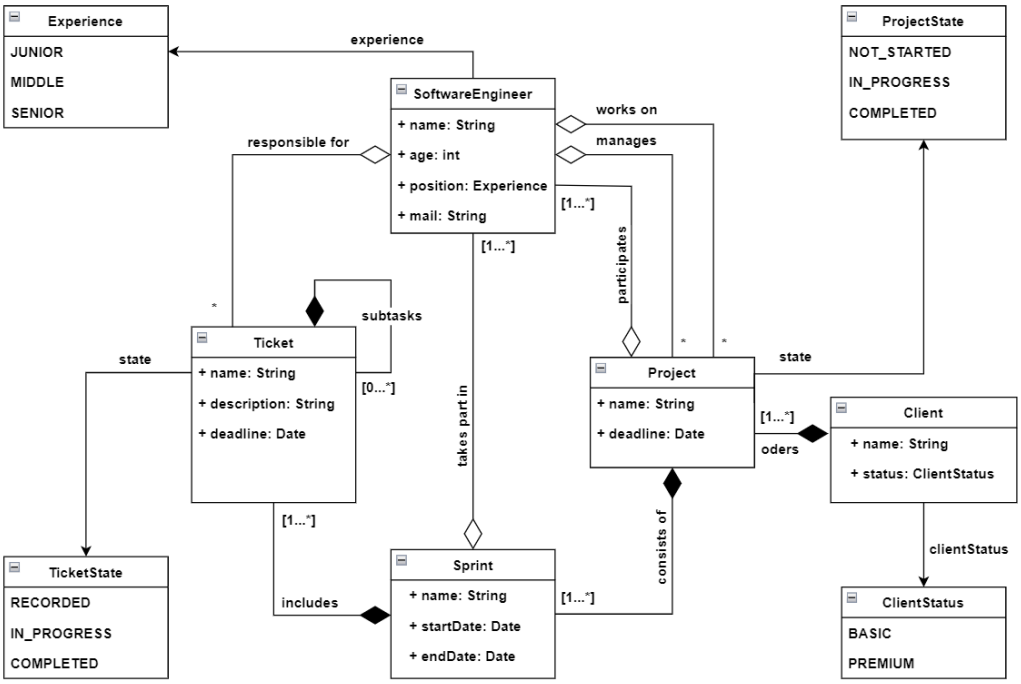


Figure 3.1: Elaborated domain model of a project management system

- ***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.
- ***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 tickets. Their ability to manage a project depends on experience.
- ***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 mostly sufficient to specify the structural requirements of a domain. By looking only at the defined entity model, some questions might already arise regarding the functionality of a system. For instance, as of right now, it is already unclear what particular 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

№	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(s) 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

associations can be regarded as two independent categories, whereas more complex constraints would 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.

Because of the infinite number of potential requirements that an end-user might specify for a domain, one of the consequences is that some of them might be unintentionally overlooked 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].

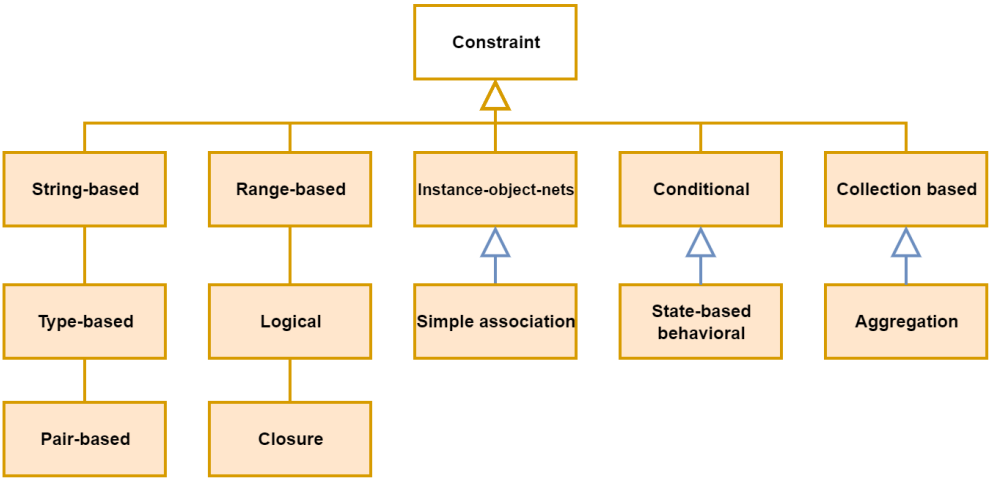


Figure 3.2: Constraint types specification

Below, each type of constraint is described in more detail. Additionally, we illustrate the relationship between a constraint classification and its relevance in specifying the previously mentioned requirements for the project management system.

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: №5, №6, №7

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

      Relevant for requirements: №1, №10

   d) ***Not null or empty*** - enforces an attribute's value be present in the definition of an entity and ensures that this value is not empty

      Relevant for requirements: №3, №11

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, №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: №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) ***Equals*** - enforces the value of an attribute to be equal to a given constant value.

      Relevant for requirements: №9

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

      Relevant for requirements: №4, №18

   c) ***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 and then applies a lambda constraint function on them. 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. 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, №9, №12, №13, №19

9. 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. Since this constraint type has a condition that depends on the system's state, this category can be regarded as a subclass of conditional constraints.

   Relevant for requirements: №13

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 resulted collection.

       Relevant for requirements: №10

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

       Relevant for requirements: №15

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

       Relevant for requirements: №10, №13, №15, №18

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: №15, №16

Requirement №	Formulating via constraint types
№1	String-based
№2	Conditional, string-based
№3	Conditional, string-based
№4	Range-based
№5	Type-based or string-based
№6	String-based
№7	String-based
№8	Range-based, string-based
№9	Range-based, conditional
№10	Simple association, collection-based, string-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, collection-based
№19	Instance-object-nets, conditional, pair-based

Table 3.2: Relation between requirements and constraint classifications

b) *Or* - the constraint is satisfied if at least one of the nested requirements is satisfied.
Relevant for requirements: №11

Considering all mentioned above, table 3.2 depicts how every requirement defined for a sample project management system could be formulated via the defined constraint classifications. Because of the rich scope of available constraint functions, some of the requirements might be formulated in different ways by applying other constraint paths.

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 adaptive 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 constraint modeling framework be integrated into other software systems?
7. How could constraints be constructed by end-users, and how could a constraint modeling framework facilitate this process?

Taking into account the questions raised above, the following subchapters elaborate on the factors of system behavior at distinct levels of interaction. Apart from it, an additional focus is placed on guaranteeing 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 while integrating new functionality. At first, an end-user expresses structural prerequisites by creating an entity relationship diagram (*Create domain*). In the next step, additional requirements are formulated by constraining the predefined environment (*Create constraints*). Before instantiating parts of a domain, 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*):

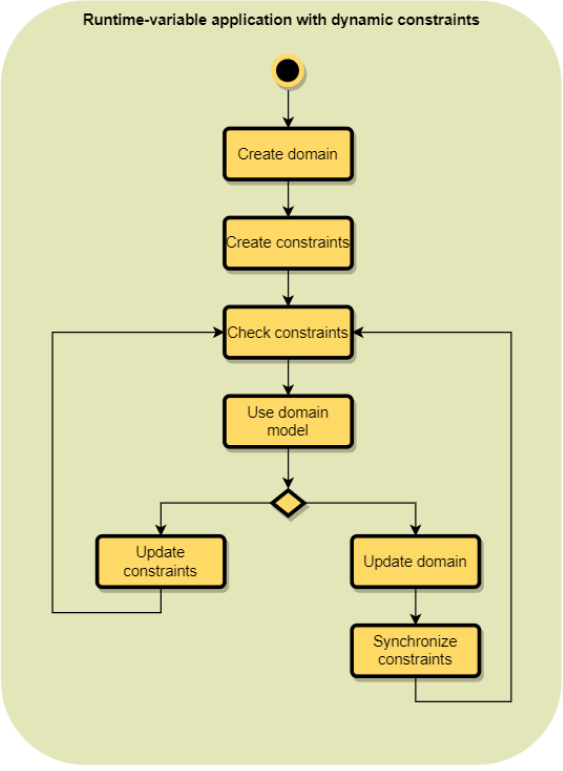


Figure 3.3: Abstract runtime model lifecycle with constraint support

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*) must take place.

These steps repeat cyclically and happen during system operation. Further activity diagram focus on constraint integrity and its life cycle in a more fine-grained manner.

3.3.2 Constraint life-cycle

Figure 3.4 depicts various use-cases for 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 explained.

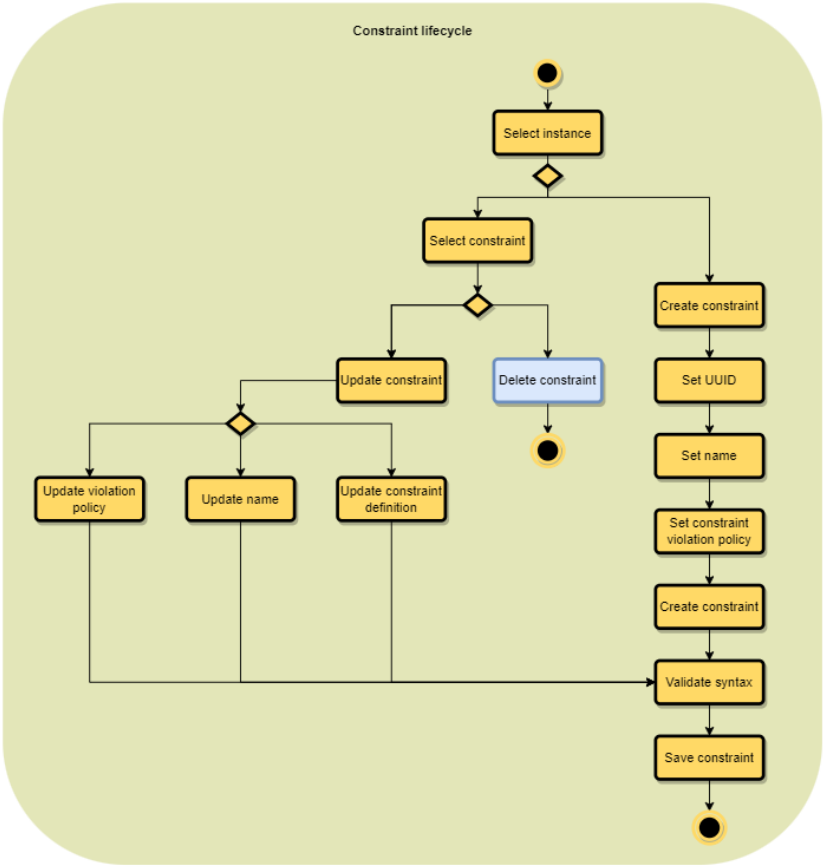


Figure 3.4: Constraint life cycle

- ***Create constraint*** A constraint is regarded as being successfully created if it retains four containing parts:
  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 functioning of the whole model. Therefore, to classify constraints by the severity of their violation, this parameter must be set either upon every constraint definition or globally in configuration by default
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 how to provide the integrity of a domain model and enforce constraint evaluation 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 verification of the constraint's integrity, analysis of affected indirect constraint, and persisting changes if all the defined restricting rules are satisfied.

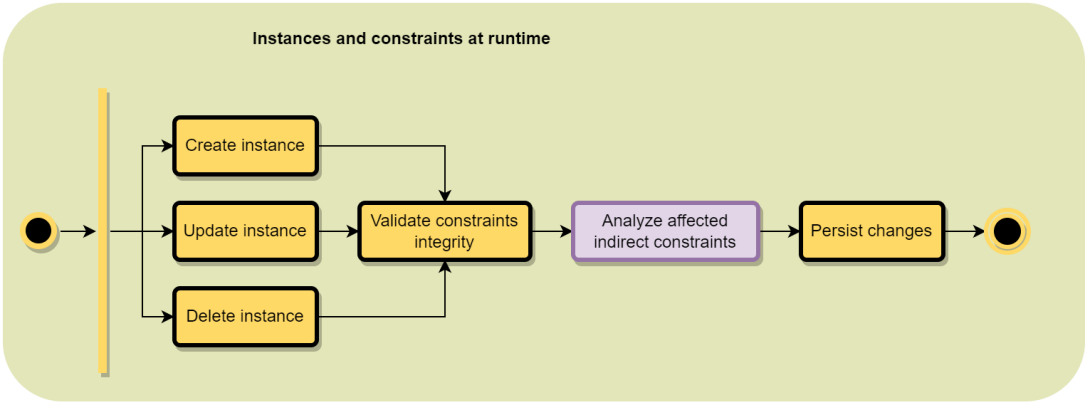


Figure 3.5: Model instantiation and constraints integrity. Abstract view.

The *Analyze affected indirect constraints* step is colored differently to highlight the necessity in terms of validating the integrity of non-structural requirements while a target domain evolves. 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 promotion 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:

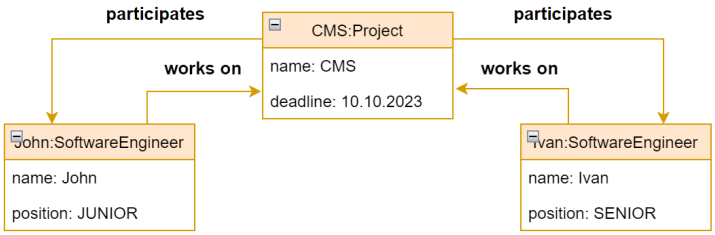


Figure 3.6: Project management system. Instance space.

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.

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. *Mantaining 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 model 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 might have encapsulated constraints or they should be pulled externally (*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*).

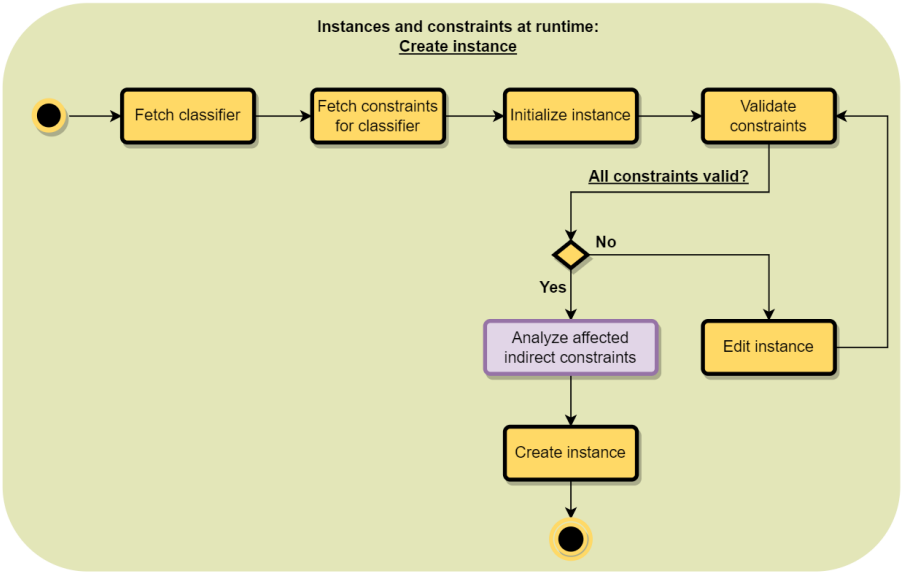


Figure 3.7: Model instantiation and constraints integrity. Create instance.

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 or an external source (*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*).

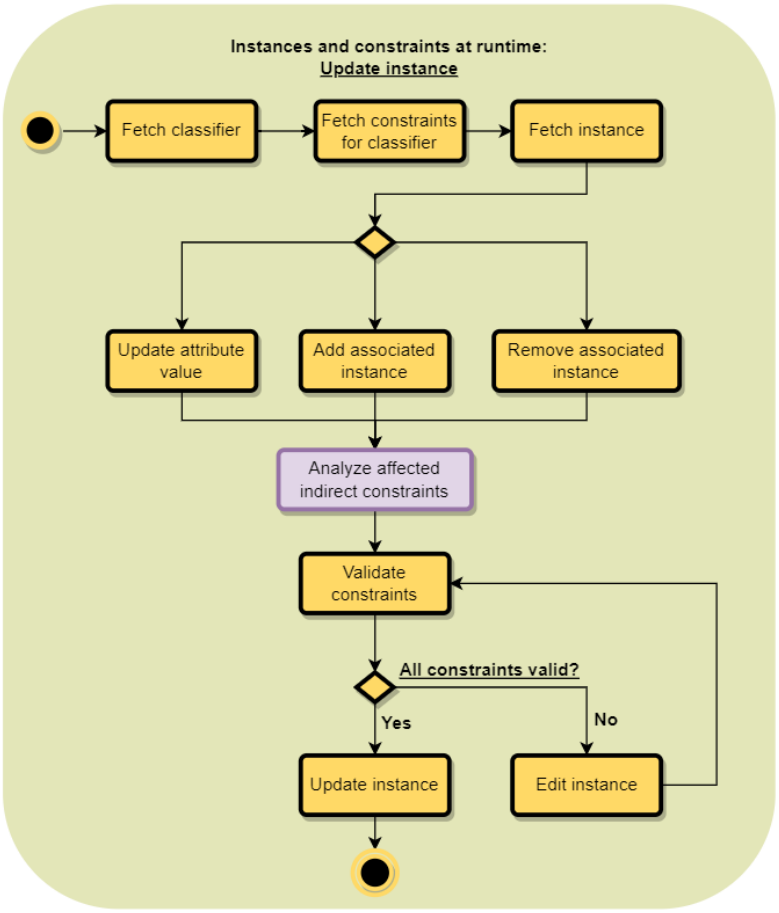


Figure 3.8: Model instantiation and constraints integrity. Update instance.

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 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*).

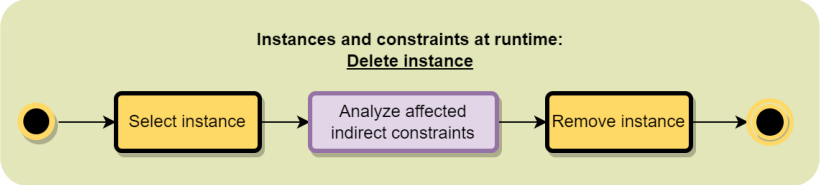


Figure 3.9: Model instantiation and constraints integrity. Delete instance.

3.3.4 Model evolution and constraints at runtime

The previous section focused on specifying requirements for constraint integrity in the context 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 adjusted automatically and keep those operations 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 or 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 did not exist. 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 3.10-3.13 elaborate on it in more detail by explaining how constraint integrity is provided for every type of domain change at runtime.

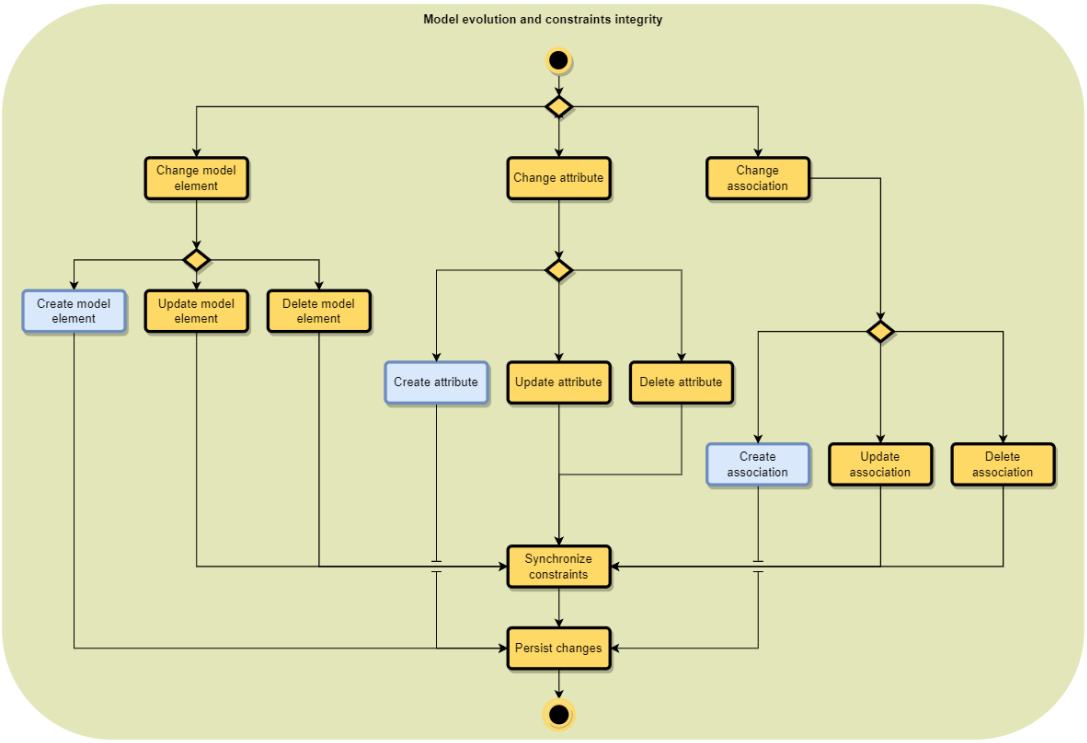


Figure 3.10: Model evolution and constraints integrity. General overview.

Model element evolution and constraints at runtime

Figure 3.11 depicts the required actions a system must undergo to ensure the correctness of specified constraints. 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*). Therefore, the process of updating all the invalid rules should be delegated 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.

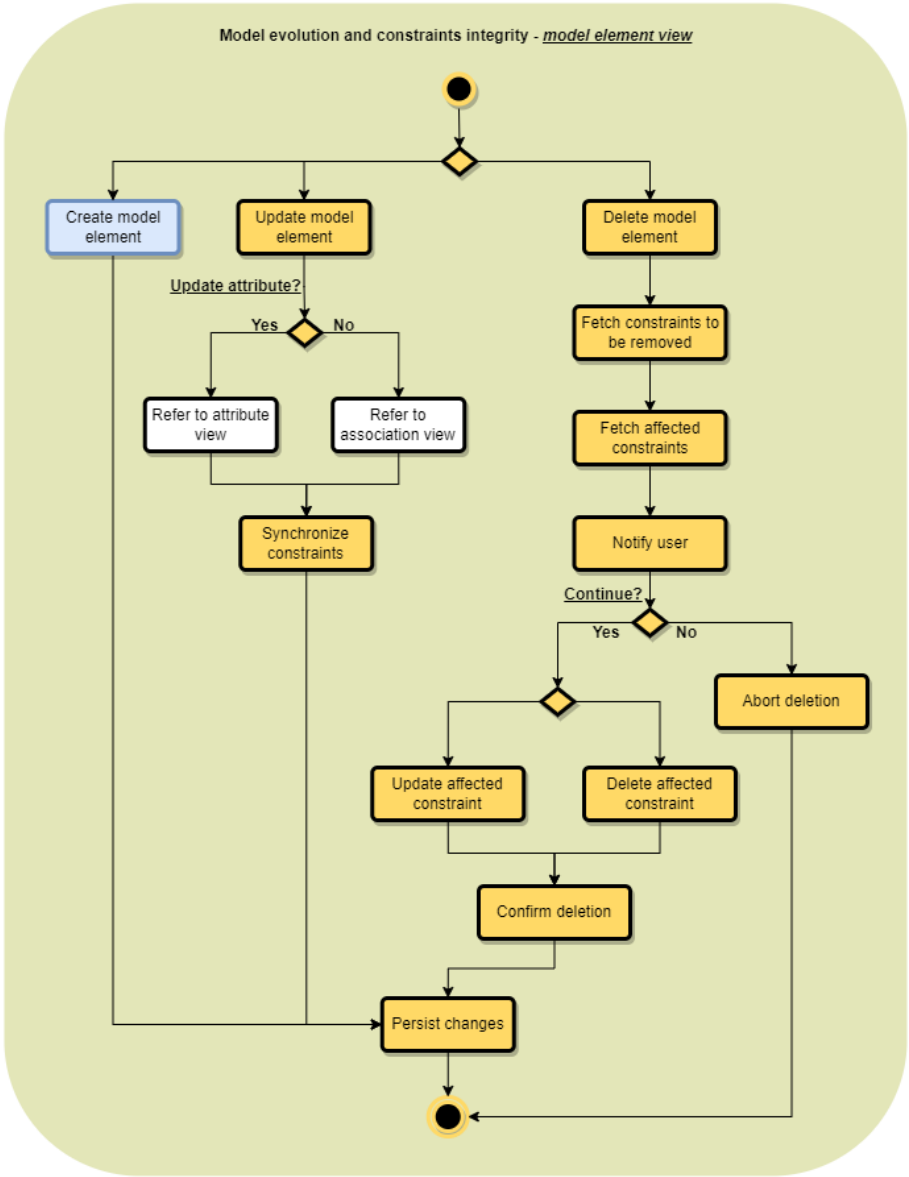


Figure 3.11: Model element evolution and constraints integrity.

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 to check 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 that refer to this attribute 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 must happen with the help of a user. 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.

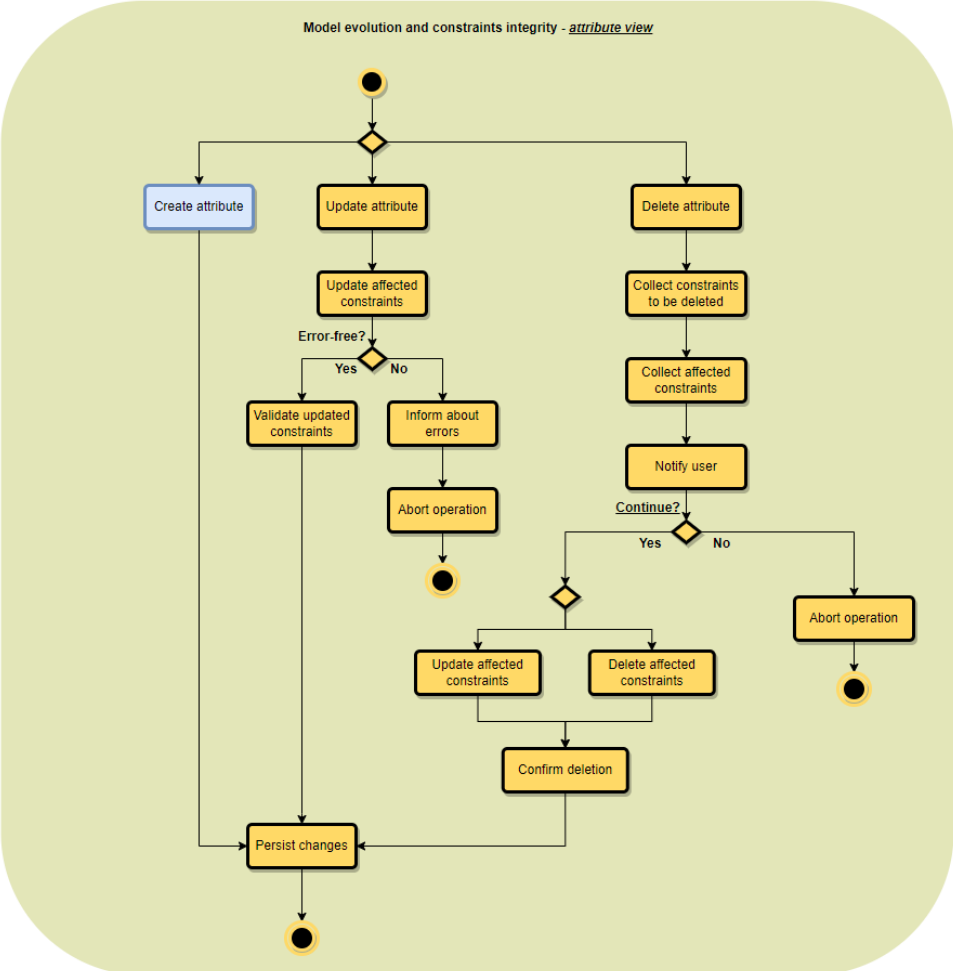


Figure 3.12: Attribute-related changes and constraints integrity.

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 behavior of a system to manage associations and enforce integrity mechanisms for constraints is shown in figure 3.13.

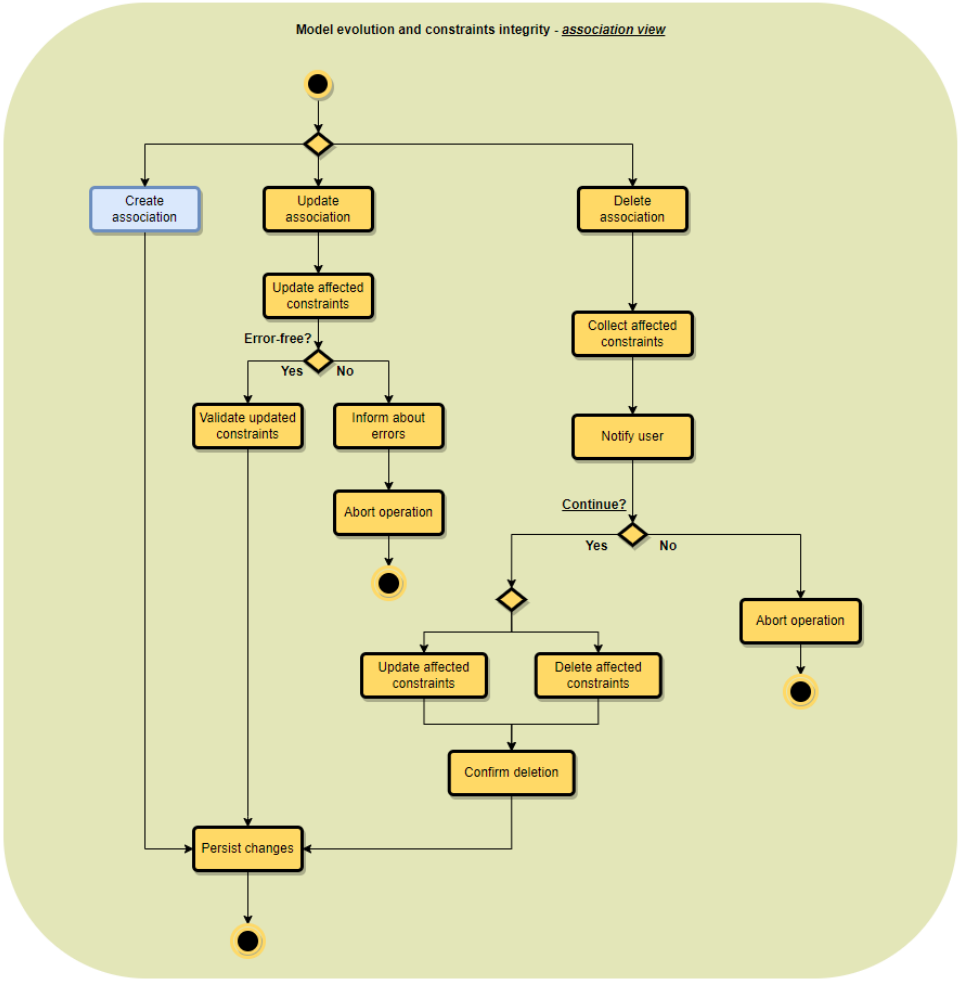


Figure 3.13: Associated-related changes and constraints integrity.

3.3.5 Constraints in the context of deep models

In the two-layered instantiation hierarchy, the constraints are formulated on the classifier level and evaluated at the time of instantiation. However, in some cases, it is complex or unnatural to present a domain element within such a flat hierarchy. To address this limitation, deep modeling utilizes the concept of a "clabject" to create entity hierarchies that span multiple meta-levels, thus blurring the traditional distinction between a type and an instance [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 a *Software Engineer* element is located on the bottom level of an ontological hierarchy and combines the traits of all its parent entities. In a world where a structural domain is tuned by additional requirements, constraining deep models is no exception. However, considering

the problematic nature of infinite meta-levels in multilevel modeling, the following questions emerge for constraining such hierarchies:

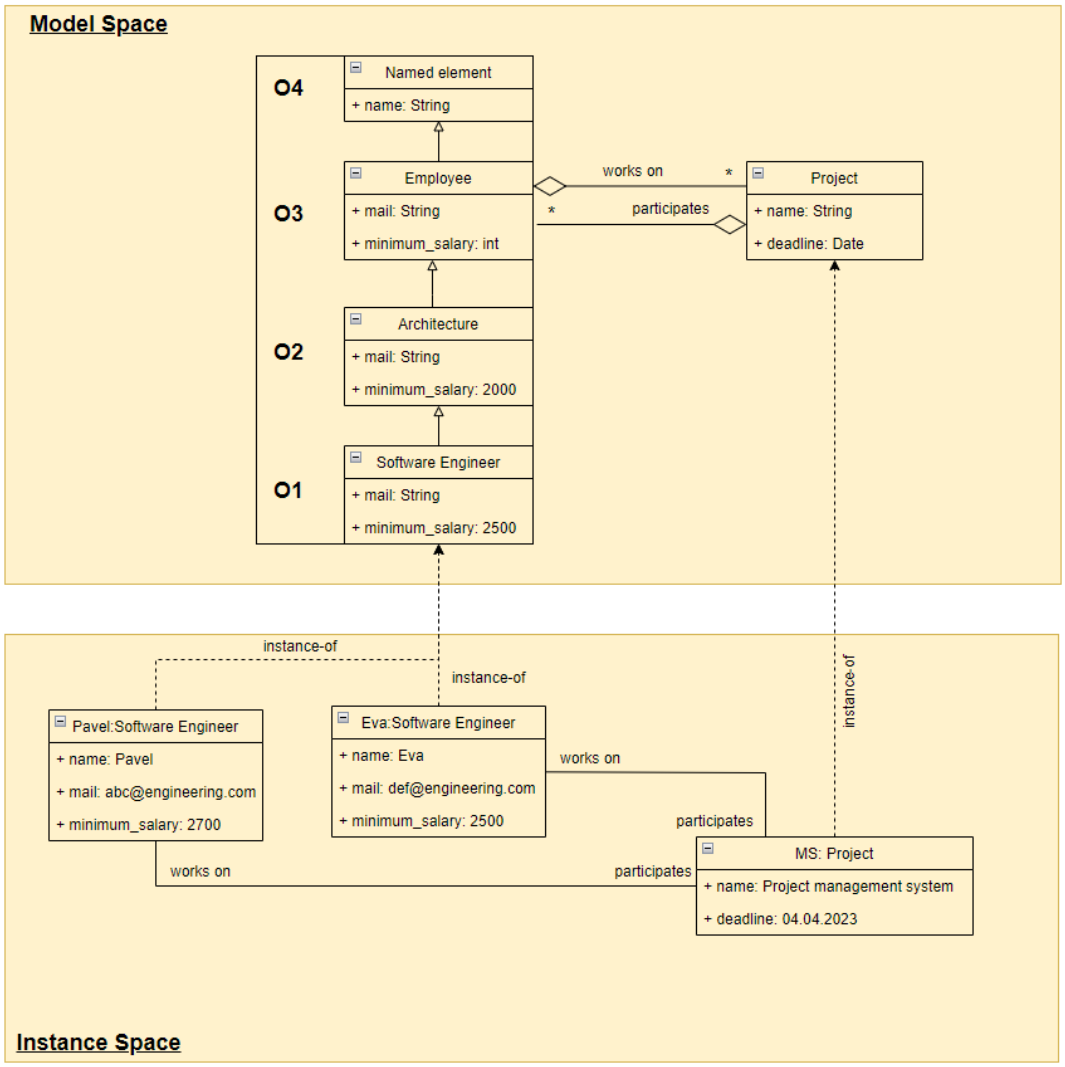


Figure 3.14: Deep modeling hierarchy - modeling and instance space.

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?
3. Should a constraint engine be aware of a deep modeling hierarchy and be able to deal with them, or could such hierarchy 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, constraints should be formulated at the types of modeling elements [CG12]. Since every clabject has traits of a classifier, a constraint engine must be able to define constraints on any ontological level of a deep modeling hierarchy. Besides, according to the deep instantiation mechanisms, all the constraints defined on the levels X, X+1, and X+n must 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 process of constraint validation occurs at the time of instantiation. However, each clabject in a model space had instance traits and could 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 while creating an object at instance space. The latter also implies that if the violation of at least one constraint during instantiation is detected, then instantiation must be aborted with the respective notification delivered to an end-user.

In the end, it is vital to consider the boundaries of a deep modeling hierarchy a constraint engine should be aware of. Suppose the support for deep modeling is embedded in the constraint engine. In that case, it increases the learning curve for 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. Nevertheless, to stay concept-agnostic and favor the general flexibility of the 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. For instance, flattening a deep clabject of *SoftwareEngineer* would result in the following structure shown in figure 3.15. It is worth noting that this is only a reference structure, and all the details are discarded here.

3.3.6 Technology independent API for domain constraints

Evaluating constraints is impossible without a reasoner that checks them against concrete instances. The abundance of different query and constraint frameworks eliminates the need to implement a custom constraint validator. Despite some similarities, each technology varies in terms of its foundational concepts, rules, use cases, and specific syntax. Therefore, the integration issue must be considered first before analyzing existing frameworks that tackle this problem. A straightforward approach is to find the best constraint or query-related framework meeting all the preconditions and use it as a logical (reasoning) layer. However, the simplicity at first glance might not pay off in a more prolonged sense. What happens if a 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 objective of constructing the specification for the constraint engine is to establish a standardized framework that can be implemented using any appropriate technology. Moreover, it enables the abstraction of the system by omitting the implementation details and not sticking to concrete technology. Therefore, by discarding the concrete-implementation-first approach, it is possible to seamlessly switch between constraint-related technology. Finally, a well-defined specification allows the gradual enhancement of a system [Kru92]. Initially, a minor prototype might be implemented following the predefined contract. However, gradually it can be updated and extended, resolving potential bottlenecks and drawbacks.

Drawing an analogy with already existing standards that are widely used, Java Persistence API (JPA) [KSK11] is a specification that allows managing and persisting data between class

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 Software Engineer clabject

entities and relational databases. JPA, by itself, does not implement any concrete operations. It is just a contract that all object-relational-mapper (ORM) tools like Hibernate or Toplink follow to implement the persistence level of a system. In case of needing to switch to another ORM, it could be done with less effort since all calls and interactions are executed through specification classes.

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 focuses on non-functional requirements that must be taken into account in a constraint modeling system driven by users. It is not a new issue to find a way that would allow end-users without technical expertise to interactively create constraints on a model. In fact, many research studies focusing on modeling like [TG07; Ber+10; YA16] claim that the constraint definition in a plain text format is unsuitable for users with no software background. One of the reasons is the steep learning curve needed to obtain the necessary programming skills and familiarize yourself with the syntax of a constraint language.

On the other hand, 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. This method of formulating constraints would be comprehensible to any human being. [HFJ11]. However, as pointed out by [KB07], the real advantage of using natural lan-

guages comes with the close integration of it to a concrete domain. However, it is challenging to achieve it 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 a way to visualize the process of a constraint specification. As summarized in the paper of [Rus+08], one of the following types of visual query systems may be considered as an alternative to natural language processing:

- **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 has a set of pre-rendered 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 clear need to weigh all the pros and cons to choose a proper tool that would allow end-users to formulate constraints regardless of their technical level of expertise. This work will partially address this problem by creating a general specification for constraint definition. However, since the primary focus of this thesis is not related to user interface problems, implementing a visual component is deferred to future work.

3.3.8 Summary

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

Functional requirements

- **Comprehensive support for various constraint types.** One of the key traits in describing 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 applying new constraints at runtime.** The scope of rules 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 constraints.
- **Support for constraints validity and evaluation during instantiation.** The system must validate constraints during a model element instantiation. If any instance-related operation affects complex object-nets constraints, the system must analyze those constraints

to keep the integrity of the whole domain up-to-date. If the latter is impossible, the system must delegate this job explicitly to an end-user.

- Support for keeping constraints meaningful during the domain evolution. Changes within a domain might affect the validity syntax of defined constraints. The final implementation must be able to synchronize the definition of affected constraints automatically. If the system performs any operations that involve deleting model elements, it is necessary to inform the end-user and allow them to determine how to proceed.
- Finite support for deep modeling. The system must accept the flattened hierarchy of a multilevel modeling element. The constraints must be checked during instantiation. Additionally, any constraints that exist within a deep clabject hierarchy must be included into the final flattened object.

Non-functional requirements

- Technology-independent API. The specification for a user-driven constraint modeling framework must be designed and implemented independently from any specific platforms to encourage reusability and facilitate maintenance.
- Constraint definition and composition. The system must provide mechanisms to define constraints and combine primitive constraints into more complex ones in a clear and understandable way for end-users. In addition, some hints for constraint definitions must be included to facilitate the process of constraint creation.
- 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.

4 Related work

This chapter evaluates state-of-the-art constraint and query engines as well as respective technologies referenced by the research community. The examined tools are assessed regarding their suitability to enforce constraints on adaptive entity models at runtime. Ease of use of technology, simplicity of development and deployment without the need to configure intricate infrastructure, as well as extensive documentation are important criteria while evaluating related works. The final goal of this chapter is to select the two most-relevant techniques according to the requirements formulated in the previous chapter that would act as the basis for implementing a user-driven constraint modeling tool.

4.1 OCL

A great variety of different OCL tools are available as open-source and commercial products. Naming just a few of them includes *Eclipse OCL*, *Dresden OCL*, *Kent OCL*, *USE*, and many others. Instead of focusing selectively on every tool, the general OMG [Gro18] specification of OCL will be considered, which all of the aforementioned tools follow to a varying degree.

According to OMG, OCL is a textual language that allows defining requirements for a model in the form of constraints. The need for such a language emerged due to the natural limitation of UML diagrams that are not expressive enough to define all the aspects of a domain. This language is suitable for design-time models and, in its original implementation, does not support semantic and syntax analysis of constraints at runtime. Therefore, the efforts to eliminate the problem of enforcing design decisions during software execution caused the creation of classical OCL extensions and the emergence of novel techniques in the research community.

In the following works of [Che+09; CA10], the authors offer a way of translating OCL constraints directly into a system using aspect-orientation mechanisms and evaluating them during execution time. Although the notion of aspect-oriented programming is powerful and it serves the needs of shifting design-time requirements to runtime, this approach does not satisfy extendability and user-driven requirements. One of the limitations is the inability to extend the existing constraints at runtime. To transform newly appeared requirements into software, the whole constraint space would have to be recompiled. Moreover, the transformation of OCL-like requirements into the AspectJ representation was not automated. As a result, it is doubtful that end-user would be able to carry out the respective transformation by themselves, meaning that a special team of developers to support such infrastructure would be required.

Another example of checking the conformance of defined constraints against a model at runtime was performed by [HR07]. This approach is based on periodic checking of the state of

a system and validating the defined requirements against its meta-model. The constraints are formulated by domain experts. An explorer occasionally takes snapshots of a system and passes the state to a constraint checker. The constraint checker evaluates and decides whether the current state of a system satisfies the prerequisites defined by domain experts. Compared to the previous approach, a strong advantage of this work is the ability to formulate additional constraints at runtime after a system is deployed and in the operational stage. Although this solution meets many requirements for user-driven constraint modeling and embodies important concepts on how a constraint-checking module should be specified in dynamic systems, it also possesses drawbacks from the performance and end-user angles. First of all, as the authors mention themselves, taking periodic snapshots of the whole system state and evaluating it against all the constraints brings performance overhead. In systems where a domain model consists of many elements, this approach causes more than significant resource consumption. Moreover, no abstraction over mere OCL syntax is made, which makes the formulation of requirements suitable only for domain specialists and leaves it challenging for non-experienced end-users.

To eliminate significant performance overhead, the evaluation of OCL can be extended by parallelizing its execution as presented in the paper of [Vaj+11]. However, when used on only one machine, a multi-threaded approach will inevitably reach its limits for parallelization due to limited computational resources. Gradually it would create the need to switch to a distributed environment that will cause other challenges of consistency and availability of such a constraint checker. Another potential solution to reduce the problem scope is to apply incremental constraint checking instead of a full reevaluation. A novel approach that uses the concept of *certificate of satisfiability* proposed by [Sán+19] allows checking only those parts of a model that are affected by changes and provides techniques for partially automatic model correction. Even though this strategy tackles the problem of model consistency in the presence of changes and seems to be promising in terms of performance, the proposed solution is aimed to be used at compile time. Therefore, before applying, it should be extended to support the dynamic nature of adaptive models at runtime.

4.2 Shacl

Shacl is a relatively recent standard language created to validate Resource Description Framework (RDF) graphs. RDF graph is a schemaless data model that consists of triple sets. Since RDF graphs have no semantics to enforce the correctness of data, other technologies must be applied to serve this purpose. A Shacl graph consists of a set of documents that shape the structure of a data graph. A Shacl validator requires two graphs as input: an RDF graph and SHACL shapes. The graph is regarded as valid if a SHACL report finds no violations [PK21]. To address our requirements regarding the expressiveness of this language and its extension abilities, we refer to the official specification of SHACL [KK17]. According to the source, shacl has an expressive standard set of constraint functions. It also has explicit support for navigation and cardinality checks. The available out-of-the-box functions are called *Core constraint components*. However, since Shacl is a simplified abstraction over Sparql [PSL08], it is also possible to extend the existing rules with custom ones using Sparql syntax. Such types of constraints that are directly derived from Sparql are known as *Sparql-based components*.

The availability of existing tools dictates the overall acceptance of technology. Even though Shacl is relatively new, it has impressive framework support. In the scope of this work, the following open-source libraries were evaluated: *TopBraid SHACL API* and *Apache Jena SHACL*. Each of these libraries allows the construction of Shacl shapes and the rule extension at runtime. Moreover, these libraries have simplified API to create RDF graphs. The proposed API is important for bridging the meta-model of a constraint engine with a web-ware technical space at runtime. Besides, numerous plugins and UI platforms are available that facilitate the def-

inition of shapes for end-users [EL16; Bon+19; Šen19]. One of these tools could potentially be integrated into a user-driven constraint modeling engine to satisfy the user-experience requirement. However, it would require further thorough analysis.

4.3 Alloy

Another modeling language that supports built-in constraint evaluation mechanisms is called Alloy. This language was officially published in 1999 by the Software Design Group of MIT [Jac04]. It has a compact syntax, and its semantics consists of sets and relations. The alloy toolkit supports a graphical interface for defining a model and its analysis. However, modeling is based on the clear Alloy syntax without simplified abstractions.

The foundation of the alloy analyzer conforms to the first-order logic. Therefore, the model evaluation is undecidable and not guaranteed to be terminated. However, it is possible to restrict the number of execution phases to a finite number. Such restriction enforces the decidability property but makes the evaluation incomplete. Hence, if the analyzer cannot find a solution for a model within a defined number of execution steps, it does not mean a model is unsatisfiable [CD03]. This property of an Alloy analyzer might imply false negative results when the analyzer claims that a model specification is valid, but it is not necessarily the case. As a result, in some cases, the evaluation of constraints might be imprecise but approximate. Nevertheless, Alloy is also actively used by the research community to fill the gap between design models and their applications at runtime [CD03; FCM18; Tik+18].

Embee [CD03] is a tool that, at its core, utilizes Alloy to analyze whether the state of a program conforms to a given specification model defined by an end-user. An end-user manually creates the specification of a model using Alloy. Then, one creates a configuration with breakpoints where the program state must be evaluated. As soon as a program reaches a breakpoint, the standalone analyzer dumps the shape of a program as well as the model specification to the Alloy evaluator. Finally, the evaluator performs the conformance checks. Apart from the presented tool, the authors also reported limitations of using Alloy as the core analyzer. First, as the authors claim, the time required for constraint evaluation increases significantly if a constraint affects more than twenty objects in one evaluation scope. Secondly, Alloy has limited support for primitive data types. Therefore, as reported, applying simple restrictions on attributes like limiting the length of a string or enforcing a string conformance to a particular regular expression pattern might be challenging. Both limitations directly affect functional and non-functional requirements defined in the previous chapter.

Another application of Alloy is to prevent failures and provide recovery mechanisms in cloud service platforms at runtime [FCM18]. The main idea is to casually connect an adaptation model that reflects the reasoner state with the running system. The variety of cloud services is unified by model variants split into dimensions. As such, the system is viewed as a set of dimensions that contain variants of a particular running service and its interrelations. The integrity of an adaptation model is satisfied by specially dedicated components that catch published events in the environment and, subsequently, migrate the state accordingly. While migrating a service, an Alloy constraint solver enforces the correctness of an adaptation model at runtime.

Moreover, Alloy found its application in the area of live modeling to provide instant feedback for end-users and instantly notify them about a model's invalid state. However, over time live modeling might trigger the drift between the model specification and its runtime state. To overcome this problem, the authors [Tik+18] suggest the automatic reconfiguration of the runtime state conforming to constraints defined by designers. The final implementation uses Alloy to specify the formal model definition, and the Alloy engine checks its conformance at runtime. The authors used Alloy to capture the structural properties of classes and the relations between them with no focus on attributes.

Finally, it is important to talk about the expressiveness of Alloy in terms of its ability to define and evaluate different types of constraints. As it was noted above, Alloy is a first-order-logic programming language. Therefore, apart from the already mentioned limitations, it is impossible to define non-first-order types of constraints. Therefore, using native Alloy makes it impossible to define aggregation or closure type constraints. [Fra+19]

4.4 GraphFrames

Another technology that could be potentially used as a core element in a user-driven modeling constraint controller is called *GraphFrames*. The below description of this technology is based on the paper of [Dav+16]. GraphFrames is a system that incorporates a mix of graph algorithms, relational queries, and pattern-matching functions. Internally, GraphFrames optimizes the execution of queries to boost performance. Implementing a constraint component with GraphFrames would involve building a graph from initial data, constructing queries for pattern matching to satisfy a model element specification, and further analyzing the output of the GraphFrames engine.

The foremost abstraction of this technology is a *GraphFrame*. A *GraphFrame* consists of 2 entity components conveyed as graph vertices whose relations are shown via edges. In addition, vertices and edges can be populated with attribute data. Figure 4.1 shows the example of a query against two constructed GraphFrames joined by the identification numbers. This query returns the number of female employees working in a given department.

```
employees
  .join(dept, employees.deptId == dept.id)
  .where(employees.gender == "female")
  .groupBy(dept.id, dept.name)
  .agg(count("name"))
```

Figure 4.1: Querying a graph with DataFrames

GraphFrames is expressive and has built-in support for comparison operators, logical expressions, arithmetic functions, and aggregations. Moreover, it supports various data types like boolean, integer, double, string, and Date. Apart from single-valued types, GraphFrames has support for maps, arrays, and other collection-based elements. Moreover, it is possible to define user-specific data types and nest existing ones to tune the graph closer to a domain. Operators and procedures can also be extended via user-defined functions by implementing a lambda interface that will be converted into a valid GraphFrames expression automatically. Unfortunately, this type of extension is only possible at compile type and does not meet our requirements for dynamic constraint extension types at runtime.

The evaluation performance of the GraphFrames engine was also conducted by [Li+21]. There, the group of Twitter engineers had to reimplement a job routine that fetches and analyzes data by defined queries. According to the outcome, the usage of GraphFrames resulted in more than thirty-three speed-up with the same datasets and input values.

Even though GraphFrames is a powerful tool that has astonishing performance on execution and supports rich semantics of operators and data types, it fails to satisfy all the requirements. Apart from its limitation for extendability at runtime, the deployment of GraphFrames has a direct dependency on ApacheSpark, which also must be installed and deployed on a machine before using *GraphFrames*.

4.5 Viatra

Viatra is a framework for reactive model transformation and incremental model querying. It is an Eclipse-based tool and is totally integrated with EMF models [Ber+08]. In the scope of related work, only the features of Viatra regarding applying and checking constraints are assessed while leaving out its model transformation capabilities.

The definition of constraints in Viatra is possible via two options. The first option is to declare rules using the DSL provided by the Viatra group [Di +19]. This definition type is shown in figure 4.2. Another option, depicted in figure 4.3, is annotation-based by using the *@Constraint* annotation [Var16]. Whenever a constraint annotation is added above a pattern definition, the code generator automatically embeds the query into an EMF model editor.

```
1  package library
2
3  import "http://se.cs.toronto.edu/mmint/MID"
4  import "http://se.cs.toronto.edu/mmint/MID/Relationship"
5
6  pattern connectedModelElems(modelElemSrc: ModelElement,
7                              modelElemTgt: ModelElement) {
8    modelElemSrc != modelElemTgt;
9    Model.modelElems(modelSrc, modelElemSrc);
10   Model.modelElems(modelTgt, modelElemTgt);
11   modelSrc != modelTgt;
12   Mapping.modelElemEndpoints.target(mapping, modelElemSrc);
13   Mapping.modelElemEndpoints.target(mapping, modelElemTgt);
14 }
15
16 pattern allConnectedModelElems(modelElemSrc: ModelElement,
17                                modelElemTgt: ModelElement) {
18   modelElemSrc != modelElemTgt;
19   Model.modelElems(modelSrc, modelElemSrc);
20   Model.modelElems(modelTgt, modelElemTgt);
21   modelSrc != modelTgt;
22   find connectedModelElems+(modelElemSrc, modelElemTgt);
23 }
```

Figure 4.2: Constraint definition using a VQL query file, taken from [Di +19]

```
1  //EMF-IncQuery pattern in the query definition file
2  @Constraint(
3    key = {"app"}, severity = "error"
4    message = "$app.id$ is not allocated but it is running",
5  )
6  pattern notAllocatedButRunning(app : ApplicationInstance) {
7    ApplicationInstance.state(app, ::Running);
8    neg find allocatedApplication(app);
9  }
10
11 private pattern allocatedApplication(app : ApplicationInstance) {
12   ApplicationInstance.allocatedTo(app, _host);
13 }
```

Figure 4.3: Constraint definition using *@Constraint* annotation, taken from [Var16]

The primary goal of using querying techniques in Viatra is to provide runtime validation of constraints and declare the domain rules to detect inconsistencies as early as possible [Var16]. Viatra offers an extensive range of available constraints like comparison operators, closure, aggregation functions, and ways to navigate through a model. It is also possible to reuse certain parts of the constraint definition by storing them in a variable. Whenever a pattern match is detected, the Viatra query engine executes a query using the provided input model [Di +19]. Having embedded the concept of incremental query evaluation, Viatra has high performance compared to other query and constraint tools. In the work of [Di +19], the authors compared the evaluation of different constraint types between OCL and Viatra. The benchmark results revealed that the evaluation time of Viatra constraints remains nearly constant, regardless of the constraint's complexity. On the other hand, the execution time of checking the OCL constraint is directly proportional to its degree of intricacy.

Setting up an incremental query engine with Viatra requires the following preliminary steps according to [Ber+08]:

1. a graph pattern that describes the structure of constraint elements must be specified. The pattern definition can include attributes, edges, and nodes itself
2. a query must be created that matches the pattern. If Viatra is able to find a pattern match, it means that the applied constraint is valid. Otherwise, a constrained model does not follow the specification.
3. once patterns and all the queries are defined, Viatra automatically generates a DSL that can be used to validate applied rules on a domain model.

In addition to examining the basic constraint and query capabilities of Viatra, its use as a constraint engine for different domains during runtime was also researched. First, Viatra was used in the work of [SVC22] as one of the components in the system to perform conformance checks of functional and non-functional requirements for runtime monitoring of cyber-physical systems and robotic applications in particular. There, the authors used Viatra to define state-based constraints that are not subject to change at runtime, like a robot's minimum and maximum battery level. More sophisticated and dynamic constraints were defined via *Complex Event Processing*, a tool for pulling data from message-oriented distributed systems [LF98]. Secondly, Viatra is successfully used in the domain of executable DSLs to accomplish monitoring capabilities and constraint satisfiability of the system's state during its execution [Ler+20]. Similarly to the previous approach, Viatra constraints were designed at compile time. Later, structural monitors are constructed from defined constraints that observe the system state for potential violations.

To conclude, even though Viatra has a significant degree of constraint types, provides navigation mechanisms through a model, and proves to be highly performant for constraint evaluation, it is not entirely apparent how to apply it in the context of a user-driven modeling constraint language at runtime. First of all, based on the reviewed papers and the tool specification, the definition of model constraints is only possible at compile time, making it hard to impose new constraints on a domain by end-users. We deem that, perhaps, it is possible to make use of Viatra for dynamic entity models at runtime, but those approaches are discouraged since they are not mentioned anywhere in the official documentation of the tool. Secondly, Viatra is a framework established on EMF with no currently available standalone version for an incremental pattern-matching engine. According to the analysis of Eclipse forums done by [Kah+16], the power of software reuse via Eclipse plugin management also implies a negative side where many users report insufficient documentation for specific software modules and difficulties with software repairment after upgrades of EMF components. Therefore, at this moment, implementing a constraint engine with Viatra would significantly complicate the process of creating a prototype based on this technology and its future support for engineers unfamiliar with the industry standards of EMF.

4.6 Gremlin Query Language

Gremlin is a graph-oriented query language created by Apache Tinkerpop. One of the main incentives for its emergence was to provide efficient and intuitive graph traversing mechanisms. As such, any Gremlin query consists of a list of declared functions and operators by iterating a graph via a user-defined path. In addition, Gremlin is an imperative query language, utilizing which end-users do not simply define graph patterns for matching but can also manage the graph traversal process by applying various functions [ŠRN16].

Apart from querying a graph, this tool supports creating, transforming, navigating, and filtering. The graph under traversal in the Gremlin ecosystem comprises two main elements - vertices and edges. The combination of vertices and their interconnections via edges is perceived as a computational unit for further processing by the Gremlin query engine [DSC16].

Expressiveness and the variety of available functions for processing can be categorized into four groups [DSC16]:

- *Branch operators* - functions that split the execution pipeline into multiple pipes for concurrent computation
- *Transform operators* - functions that take traversal objects as input and transform them into another output type
- *Filter operators* - functions that filter out unnecessary data not satisfying specification and return traversal objects according to the initially set conditions
- *Create/Update-related operators* - functions to manipulate a traversal graph by adding, updating, or removing vertices or edges from it

Moreover, Gremlin has an API to access and evaluate the meta-information of a graph, like checking the total number of graph elements or accessing the size of objects in a traversal operation. In order to make a given language adaptable for different domain purposes, this language can be extended on demand by defining custom functions and operators. [DSC16].

Being a language designed initially for querying and traversing complex graphs, Gremlin does not have native support for constraints [ŠRN16]. However, its extensive query semantics and filtering operators would assumingly allow the definition of constraint types via predefined query blocks exposed later to end-users. Moreover, a user can create a custom DSL with traversal functions that will be generated automatically by the engine. It is a strong advantage that would allow the creation of fully customized DSL targetted to constraint definition and evaluation. Another limitation of Gremlin, as claimed by [TPA12], is its inability to run a query execution by several threads concurrently. Even though the parallel query evaluation is naturally impossible with Gremlin, it still has support for functions that split processing into nested traversals and merge their results upon the completion of execution. In conclusion, the preliminary look at this technology does not violate any of our requirements and is viewed as one of the candidates for a constraint engine in the end-user modeling constraint component.

4.7 GrGen

GrGen is an open-source graph transformation system. It provides a rule-based language to perform graph transformations, which can be used in various domains, such as software models, databases, and social networks [JBG17]. To allow querying a target domain model, GrGen uses a combination of graph rewrite blocks and graph pattern matching techniques. Hence, end-users can declare rules that would guide the graph transformation and specify how specific patterns within the graph should be matched to modify a target graph. As a logical engine, GrGen provides a powerful reasoner to be both expressive and highly performant, letting end-users define complex rules that can be executed efficiently even on large graph datasets [Gei+06]. In addition to the core graph functionality, GrGen also offers a number of tools and utilities for visualizing, analyzing, and optimizing graph transformations. These include a graph *visualization tool*, a *profiling tool*, and a *rule optimization tool* [JBG17].

One interesting question in our context is whether GrGen could substitute OCl in extending a domain model with user-driven constraints. To the best of our knowledge, GrGen does not have explicit support for constraints. However, their definition and evaluation could be simulated by using patterns. A pattern is a template that defines the structure of a graph that should be matched. A pattern consists of several nodes and relations expressed via edges [JBG17]. In terms of expressiveness, GrGen allows users to define complex graph transformation rules which support a wide range of graph patterns and can perform graph manipulations at different levels of abstraction [JBK10].

GrGen is designed to be a high-performance graph transformation system [Gei+06]. One factor contributing to its efficiency is an internal graph optimization targetted for fast access and manipulation of graph elements. Moreover, graphs, rules, and patterns are compiled and stored in memory. It allows GrGen to quickly search for matches to complex patterns and apply transformations to the graph.

Even though GrGen is a powerful and flexible tool for working with graph-based data and systems, it is doubtful that it can be used in the reasoning layer of the constraint modeling framework. There are several reasons for it. First, this tool has a steep learning curve since it has its own syntax and paradigm. In addition, there are too few usage examples or articles on how this tool could be used in practice. Secondly, it is impossible to state beforehand whether this tool is feasible to constrain adaptive models because it is unknown whether it could formulate all required constraint types. The reason for it is that GrGen is primarily designed to be a graph-transformation system, and its meta-model does not know the definition of a constraint. Finally, the most recent implementation of GrGen is written in C and not in Java. Although running C# code from Java would still be possible using Java Native Interface (JNI) [Cor12], it would require setting up additional configurations and components. All this would make the whole infrastructure more complicated to manage and maintain.

4.8 MetaDepth

Metadepth is a metamodeling framework created to overcome the limitations of the traditional four-layered meta-hierarchy by supporting an infinite number of meta-levels. The architecture of Metadepth is based on the MOF metamodel with some essential extensions to support the notions and techniques for deep modeling. As such, the Metadepth's metamodel defines additional types like *Clabject* and *Potency*. *Clabject* represents the dual facet of a modeling element to be regarded simultaneously as a type and an instance. *Potency* restricts the available levels across which a modeling unit can be instantiated [DG10].

To maintain the conformance of deep models against domain-related specifications, Metadepth supports the declaration of constraints that are defined in the context of clabjects. The task of a modeler is to provide the constraint potency and operations on which the constraint is evaluated, like update, delete, and so on. The syntax and expressive capabilities of the Metadepth constraint engine are similar to OCL capabilities since the Metadepth constraint engine is an extension of it. Constraint declaration is possible via Java and EOL [KPP06]. Figure 4.4 defines a sample constraint definition in Metadepth [DG10].

```
1 Model Store@2 {
2  Node ProductType@2 {
3   VAT@1      : double = 7.5;
4   price@2    : double = 10;
5   discount@2: double = 0;
6   minVat@1   : $self.VAT>0$
7   minPrice@2: $self.price>0$
8   maxDisc@2 : $self.VAT*self.price*0.01+self.price<self.discount$
9   /finalPrice@2:double=$self.VAT*self.price/100+self.price-self.discount$;
10  }
11 }
```

Figure 4.4: Metadepth constraint definition sample, taken from [DG10]

The definition of constraints can be clabject-specific as well as span some deep subhierarchy of a domain. For instance, a constraint with an assigned potency of one will be checked on the next instantiation, i.e., only once. In contrast, the same constraint with a higher potency will be evaluated across the number of meta-levels equal to the given potency. Moreover, Metadepth also supports reusing constraints via its definition on a model level and repetitive reassigning to different modeling elements [LGC15].

The value of the potency of every clabject must be one lower than that of its direct type

context *Clabject*
getDirectTypes() implies forAll(t | t.potency = potency+1)

(a) Linguistic constraint sample

The value of a ProductType's price attribute has to be smaller than or equal to its RRP attribute

origin*(1) ProductType*
self.price <= self.RRP

(b) Ontological constraint sample

The (default) prices for instances of CarType and their instances must be greater than zero

origin*(1..2) CarType*
(a) self.level < 2 implies self.price > 0
(b) self._l_.level < 2 implies self._o_.price > 0
(c) self._l_.level < 2 implies self.price > 0
(d) self.^level < 2 implies self.price > 0

(c) Hybrid constraint sample

Figure 4.5: Metadepth constraint types, taken from [AGK15]

Metadepth also has a rich scope of attributes that could be referenced and limited by the specification of its constraint language. More specifically, a constraint in Metadepth can address metamodeling concepts of the language (*linguistic dimension*), end-user-defined elements and domain (*ontological dimension*), or the combination of every type. Figure 4.5 highlights the different scopes of constraint applicability in Metadepth [AGK15].

To conclude, the targetted audience of Metadepth is the experts who need to create models moving beyond the classical MDE infrastructure and validate them. In addition, the extensive support for deep modeling makes it a perfect tool for complex systems that do not function fluently in a traditional meta-hierarchy space. However, embedding the constraint component from Metadepth into our constraint modeling engine would bring more overhead in terms of its use by end-users unfamiliar with the notion of deep modeling. Moreover, another pitfall is its current maintenance state and support. According to the official website of Metadepth [LGC10], the last release was in 2018, and no further announcements on adding new functionality or fixing current issues have been found.

4.9 Melanee

Mealee is another level-agnostic modeling language. Compared to Metadepth, Melanee entirely relies on EMF infrastructure and consists of embeddable plugins that extend the functionality of its core module. In particular, it has a dedicated plugin [Kan14] that can be added as a dependency to the tool to constrain a domain model. This plugin extends the basic capabilities of OCL by introducing language and ontology-specific operations. In addition, Melanee has a graphical plugin that can be used to define a deep modeling domain via diagram notations instead of relying on textual definitions [Kim+16].

The architecture of Melanees is shown in figure 4.6. The core package is based on the Eclipse infrastructure and includes the following necessary modules [AG16]:

- *Linguistic model* - core definition of a deep modeling language with required concepts

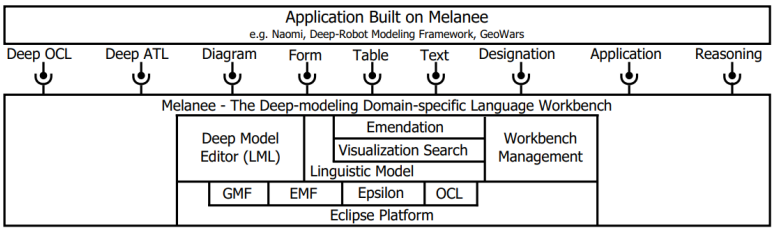


Figure 4.6: Melanee architecture, taken from [AG16]

and structure to make it level-agnostic

- *Deep model editor* - graphical interface similar to UML to define, edit, and update multi-level domain models
- *Emendation service* - service to enforce integrity mechanisms of a system in case of updates or invalidations of some specification properties. Upon detecting the drifting that triggers inconsistency, it either attempts to update the state automatically or notifies an end-user about recent updates and provides further possible suggestions to tolerate the inconsistent behavior [LA18]
- *Visualization search* - the rendering component that allows users to search and visualize a specific model element
- *Workbench management* - encompasses tools and artifacts to manage the environmental state of Melanee

Besides the core modules mentioned above, Melanee has open interface classes that could be implemented as an extension by plugins. For instance, the *Deep OCL Dialect* [Kan14] is an Eclipse-based tool developed to support the constraint definition within Melanee. As a result, the constraint module can be used to define constraints spanning several meta-levels. The constraint engine's expressiveness is equivalent to OCL. On the other hand, as pointed out by [Kan14], not all constraint types are currently implemented. Moreover, the dynamic extension of constraint types is also impossible because a user would have to implement exposed interfaces inside a program code explicitly. Finally, the created OCL dialect for deep modeling is a research prototype with some performance bottlenecks that might contradict our non-functional requirements.

4.10 Conclusion

This chapter analyzed various constraint and query languages and their viable application in the context of model adaptability at runtime. In addition to reviewing the capabilities of each constraint language, multiple technologies have been evaluated that exploit domain constraining as part of a system's component to guarantee the correctness of a model according to an end-user-defined specification. The additional evaluation of the research-oriented techniques with one or another constraint language was necessary to gain an in-depth understanding of its capabilities. As a result, state-of-the-art constraint languages were evaluated based on their compliance with the requirements outlined in the previous chapter. The summary of the related work analysis is depicted in Table 4.1.

The mapping between the criteria abbreviations and their actual meaning is the following:

- ***T-1*** - High language expresiveness. Can a tool be used to express all stated constraints?

Tool	T-1	T-2	T-3	T-4	T-5	T-6	T-7	T-8	Total
Gremlin QL	1	1	1	0.5	1	1	1	1	7.5
SHACL	1	1	1	0.5	0.5	1	1	1	7
Alloy	0.5	1	1	1	0	1	1	1	6.5
OCL	1	0.5	0.5	0.5	0.5	1	1	1	6
Viatra	1	0.5	0	1	0.5	1	1	1	6
GrGen	1	1	1	1	0	0	0.5	1	5.5
MetaDepth	1	0.5	1	1	0	1	0	1	5.5
GrahFrames	1	0	0	1	0.5	1	0.5	1	5
Melanee	1	0.5	0	1	0	1	0	1	4.5

Table 4.1: Comparison of constraints tools at a glance

- *T-2* - Support for dynamic extendability. Is it possible to extend the scope of static constraint at runtime?
- *T-3* - Standalone tool with no inter-dependencies. Is it a self-contained tool with no transitive dependencies to other software components?
- *T-4* - Decent performance. Is there any negative feedback in the research literature that criticizes the performance of a reviewed tool?
- *T-5* - Having low entry-level and being thoroughly documented. How easy is a tool to learn and use? Is the documentation clear and comprehensive?
- *T-6* - Compatibility. Is the tool compatible with Java, or is it cross-platform so that one can use it on multiple operating systems?
- *T-7* - Community and support. Does the tool have an active community of users and contributors?
- *T-8* - Cost and licensing. Is it freely available, or does it require payment to access and use?

It is worth noting that constraint languages were evaluated only for a subset of the requirements. Requirements such as constraint lifecycle or its validity and integrity during domain evolution were currently neglected. The reason is that these prerequisites do not directly apply to a particular constraint language but to the system as a whole. Every declared criterion is graded against a selected constraint language depending on its requirements satisfiability. As such, one point is allocated for full requirement conformance. Half of the point is credited if a requirement can only be met partially. Finally, no point is given if the requirements cannot be fulfilled. In addition, it is worth mentioning that to evaluate the performance aspect, those tools were not compared to each other to determine what technology behaves faster or slower in some environments. In fact, it would be difficult and inaccurate to compare performance among the researched tools since many of them are designed for different types of graph-based modeling and analysis tasks. Instead, to provide some remarks about performance, we look for some negative feedback in the literature regarding an assessed tool. Since this criterion is not accurate, performance is not used as a decisive factor but serves as a reference.

Such tools as *Melanee*, *GraphFrames*, *MetaDepth*, and *GrGen* scored the lowest according to the specified requirements. One of the common flaws of these tools is their limited community and support. It might lead to difficulties in updates and bug fixes. In addition, it causes limited resources for troubleshooting to resolve some issues that a software engineering team might

encounter. Another difficulty in integrating one of them into the future constraint engine is their high entry-level or insufficient documentation, which would hinder the implementation process and further maintenance. Another group of tools comprising *OCL*, *Alloy*, and *Viatra* has no overarching reason not to fulfill the requirements. However, the limitations are related to infrastructural inter-dependencies, limited ability for dynamic extension, or their non-trivial usage. Finally, *SHACL* and *Gremlin QL* achieved the highest values satisfying all the defined requirements related to a constraint modeling language. Therefore, both will be integrated into the user-driven constraint modeling framework by implementing exposed interfaces of the standard specification. The following chapter will propose the unified specification as well as elaborate on design decisions, architecture, and how the selected tools can be encapsulated into the framework. Subsequently, each implementation will be tested and compared in the *Case Study* section.

5 Architecture and design decisions

After familiarizing a reader with the necessary foundation information, establishing system requirements, and researching cutting-edge tools and techniques to constrain adaptive entities, the subsequent chapter focuses on the architecture and design aspects. In particular, this chapter establishes the specification enabling a user-driven constraint modeling framework to be integrated into modern software systems. First, the metamodel of a constraint modeling framework is specified. Secondly, the general workflow of software interaction with the tool is presented. Next, this chapter introduces mapping techniques and required services to enable end-users to apply constraints and check them at runtime. Finally, the integrity mechanisms are explained, taking into account the high dynamism of adaptive models.

5.1 Domain of constraint and constraint functions

A specification clearly defines the requirements and expectations of a particular software system. Without a well-established specification, the concrete implementation might be inaccurate or incomplete, leading to unexpected results or inconsistencies. This section focuses on the domain layer that should be part of the user-driven constraint modeling specification. To clearly define the boundaries of the framework, we present the core entities of the domain layer and what role they play in the overall picture.

Domain layer of models

To allow the creation of constraints, the framework should know the semantics of a model element that will be used as a context for constraint definition. Since the framework is unaware of the internal modeling structure of client software, it is necessary to map the model-ware space between the constraint modeling framework and target software. In order to achieve it, the specification defines its own metamodel for model space. This part of the domain is presented in figure 5.1 and explained in more detail below.

As such, *ModelElement* has a name and an identifier attribute. Moreover, it contains references to *Association* and *Attribute* entities. Each *Association* links two domain entities by particular relation. In addition, an *Association* stores the path between two connected entities. This is required to allow navigation in a model graph while evaluating constraints. An *Attribute* refers to a characteristic of an entity. It contains a key and a datatype that shape a constructed element. Moreover, every *Attribute* used in a constraint must have an *attribute* parameter matching a specific pattern. If a constraining attribute is beyond a context entity, it is crucial to indicate how it can be located by setting up *navigation*.

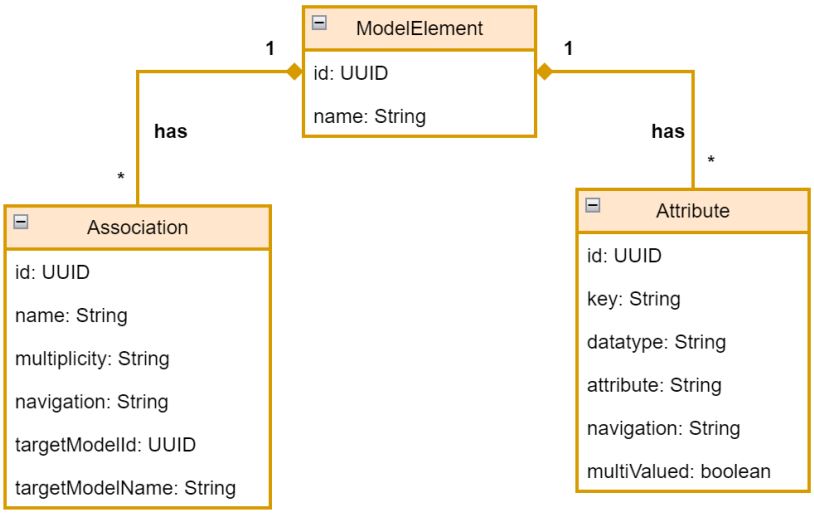


Figure 5.1: Domain layer of model space

Domain layer of instances

Evaluation of constraints happens in the context of a target instance. If an end-user wants to check a non-trivial constraint spanning multiple interrelated instances, then the relevant subgraph of all instances must be submitted. Due to the lack of information on how the graph of instances is presented in a client system, the specification exposes the platform-agnostic metamodel of an instance. Therefore, every graph used as an input source for constraint evaluation must first be mapped to the technical space of a constraint modeling specification. The respective metamodel is shown in figure 5.2.

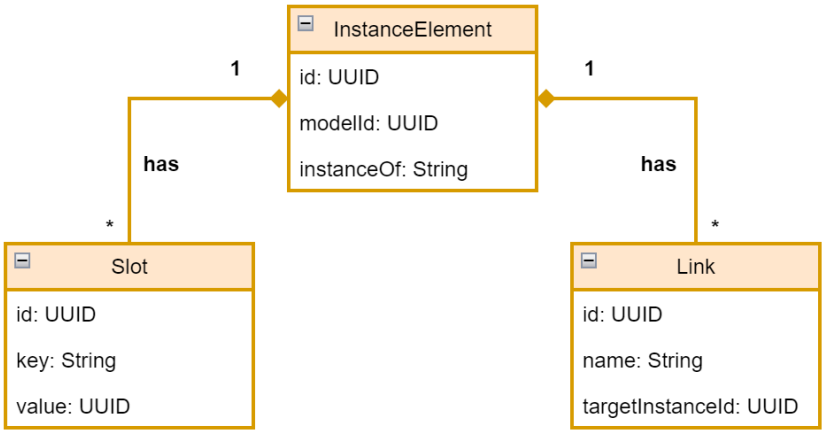


Figure 5.2: Domain layer of instance space

Every *InstanceElement* must have an identifier and reference to a model element used for instantiation. *Link* is an instance of an *Association* from the model space. It defines a connection between two definite instances linked by relation. *Slot* is an instance of an Attribute. Every *Slot* fills up an instance with exact values, thus providing a particular meaning.

Domain layer of constraints

In order to provide the capability to define constraints, the specification must declare the respective classes. Creating a constraint in our context means instantiating a *Constraint* class on the client side. The structure of a constraint is presented in figure 5.3 and has the following semantics.

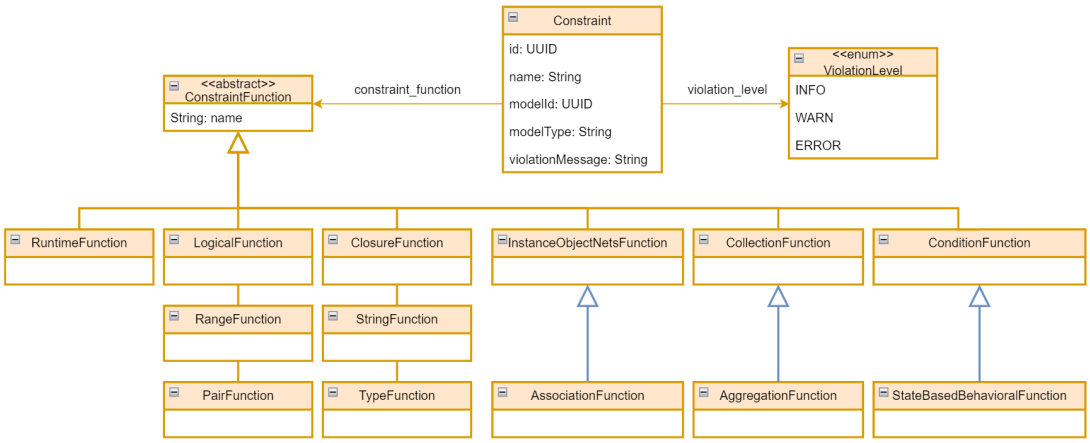


Figure 5.3: Domain layer of constraints

Every constraint must have a name and an identifier. The identifier is created automatically during instantiation. Besides, every constraint must have information about a target model element as a context. Moreover, *ViolationLevel* is designed to indicate that constraint violations might have different impacts on the client software. Upon constraint creation, end-users should decide what constraints should be strictly validated and which should carry informative meaning, not causing exceptions. Finally, every *Constraint* must have a reference to *ConstraintFunction*. A constraint function is just an abstract class with the necessary information every concrete function inherits. As observed, the specification domain implements all the constraint types defined in one of the previous chapters. Moreover, it also supports runtime functions that an end-user can define at runtime.

5.2 General architecture of a system using user-driven constraint modeling

Defining a top-level architecture is one of the most crucial steps during the design phase of the software lifecycle. It clearly explains how a client application could integrate a user-driven constraint modeling tool and how all the components should interact. Hence, the reference architecture shown in figure 5.4 will help us to guide further implementation of the system, ensuring that it stays scalable and easy to maintain.

To start with, we assume that a client application supports dynamic entity models at runtime. As an example, a client could embed any library that relies on the adaptive object model pattern or implement it themself. All the calls related to constraint definition, validation, and synchronization must go through specification methods since all interfaces and classes are declared there. Before starting an application, a client must do the following:

1. **Implement model and instance mappers.** The specification exposes classes that are used during constraint modeling. In order to bridge a client software with the framework, the mapping of those two meta-elements is required

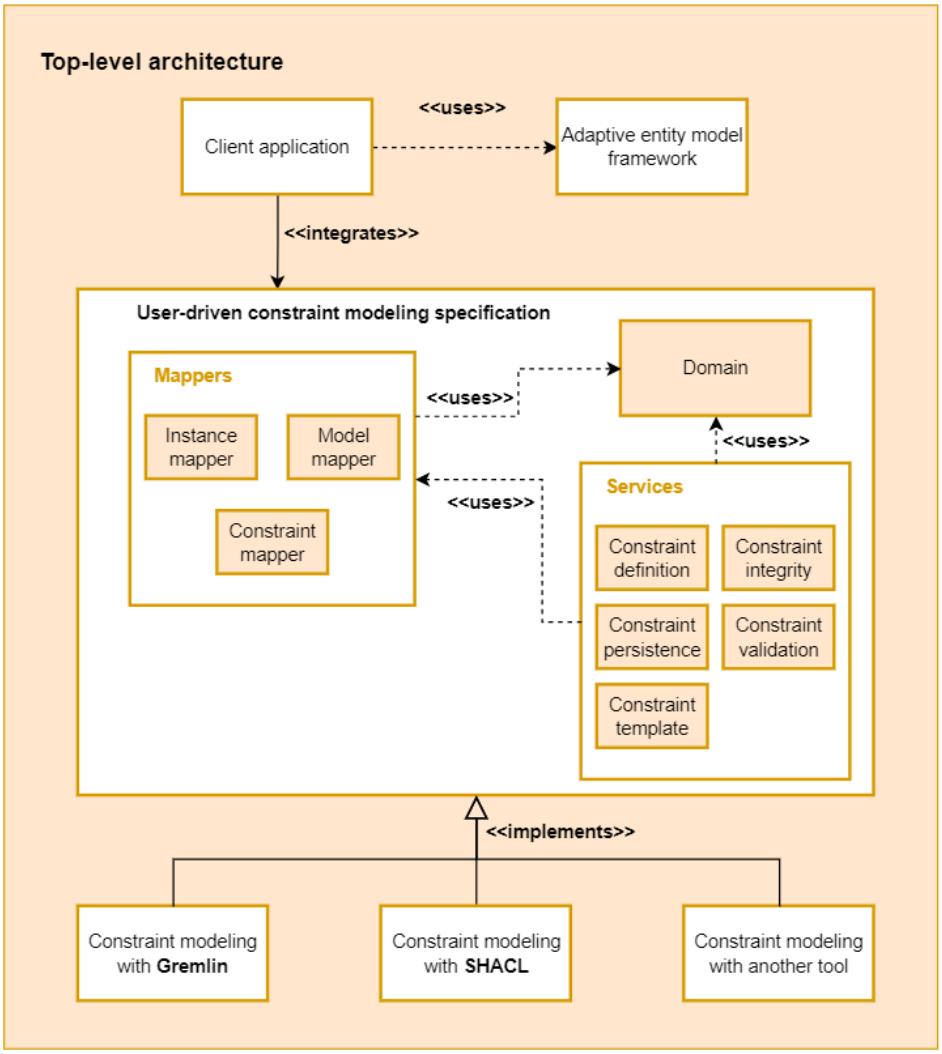


Figure 5.4: Integrating constraint modeling framework: top-level view

2. Instantiate platform-specific constraint validation service. The logic to validate constraints varies depending on the tool. As of right now, a client application might choose between SHACL and Gremlin implementations

3. Instantiate rest of the services. To make the constraint framework quick and effortless to set up, we offer pre-built implementations for all services, excluding the validation service. For instance, a client does not need to implement an interface responsible for constraint definition since its default implementation is already available. Nevertheless, to allow further customization or adaptability, clients can always extend existing services or create their own implementations

Moreover, if a client is interested in implementing their own platform-specific constraint modeling tool, they must do the following steps:

1. Implement a constraint mapper. Every defined constraint submitted for evaluation is sent to the platform-specific framework in an abstract syntax classified by the specification. It is the task of a concrete modeling implementation to transform a constraint from an abstract to a platform-specific view

2. Implements constraint validation service. The implementation of this service will vary

because it relies on mapped constraints. Hence, it should be the client's duty to provide a concrete implementation for this interface

To sum it up, the following top-level architecture shows that a client application can easily integrate the framework into its workflow by implementing a couple of mappers and configuring several service classes. Moreover, it provides clear boundaries between different classes and relations among them. A user always has a choice to use the default implementation or customize it to their business needs by adapting some parts of the specification.

5.3 Mapping technical spaces

Specification exposes a set of mappers needed to bridge diverse technical spaces. Since the specification does not know the metamodel of its concrete implementation and a client application, we construct a layer of mapper classes by design. Those classes and the way how they can be integrated are shown in figure 5.5.

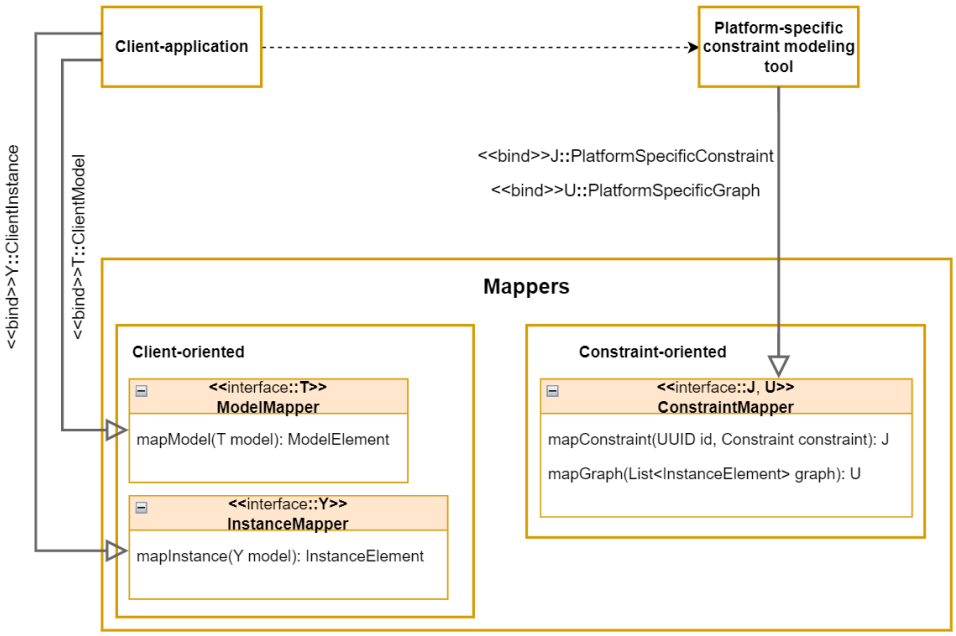


Figure 5.5: Overview of constraint specification mappers

All the mappers are designed to be generic to allow operating on different data types. Those mappers can be classified based on the specific context in which they are further utilized. As such, a client software product must implement a *ModelMapper* and *InstanceMapper* to map model and instance entities to align with the domain of the constraint framework. On the other hand, *ConstraintMapper* interface must be implemented by the constraint framework itself. This generic interface includes two methods. The first method is needed to map an abstract constraint syntax to a platform-specific one. The second method maps a graph from the abstract, specification technical space to a platform-specific one. Hence, the data transformation workflow from a client's view to a constraint framework becomes modular and flexible for adjustments. This workflow is depicted in figure 5.6.

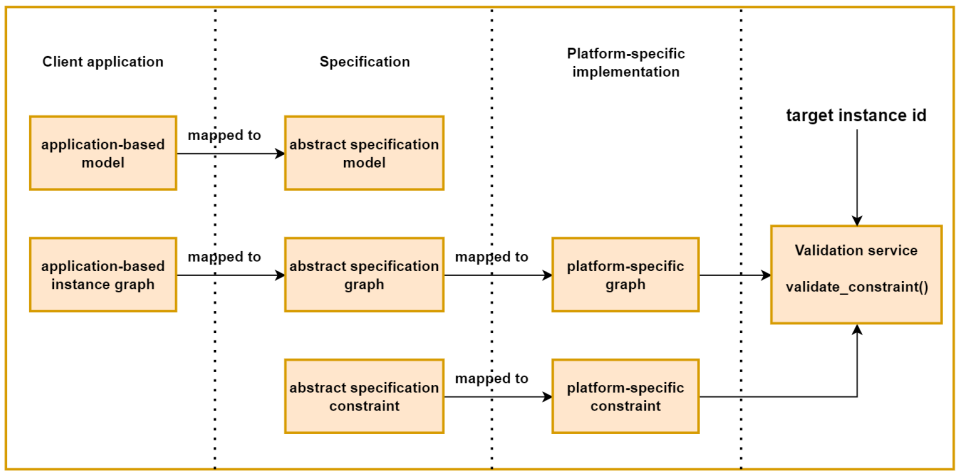


Figure 5.6: Mapping data workflow

5.4 Constraint definition and creation

The target audience of a constraint modeling framework is rather end-users with potentially no programming expertise. In this regard, it is essential to think about how the process of creating constraints could be both intuitive and flexible. Based on the previous chapters, it is clear that consumers are often discouraged from working with the raw syntax of some constraint tools. To overcome this issue, we allow users to deal with predefined constraint blocks instead of requiring them to write some programming code. As a result, end-users can modify templates to create a constraint without knowing how to write complex code. That significantly lowers an entry-level and makes it more straightforward for non-programmers to contribute to a software system.

Figure 5.7 presents a class diagram reflecting the specification parts responsible for creating constraint templates and delivering them to client software. As such, *TemplateFunctionService* is the core element of this functionality since it exposes methods to manage templates. This interface has a reference to *TemplateFunctionInitializer* . This class is a bootstrap service that creates templates for each currently supported function. To create function and constraint templates, the *TemplateFunctionInitializer* service utilizes *FunctionType* and *FunctionMetadata* classes. *FunctionType* stores all function types, whereas *FunctionMetadata* has data about each concrete function, like its name, return value, and what parameter it accepts. In addition, to allow extending static functions with newer ones while an application is running, TemplateFunctionInitializer defines the template for a runtime function. The runtime function must return a boolean value and be created by the end-user providing a platform-specific implementation code. Consequently, if a runtime function is used within a constraint, it will be dynamically parsed and executed by a platform-specific constraint engine without the need for recompilation.

By using predefined templates, end-users might easily create constraints that will adhere to the abstract syntax of specification. As mentioned above, its abstract representation must be mapped to the platform-specific constraint modeling implementation during constraint evaluation. An example of a predefined constraint is shown in figure 5.8. As can be noticed, a template consists of two parts. The first constructs a template for a constraint and fills it with default and hint values. The second part consists of a composite template for the constraint function. In this case, the template function consists of two range-based functions combined by a logical *AND* operation. An end-user just has to replace advice values with concrete and meaningful values.

As a result, providing constraint templates instead of making end-users deal with program-

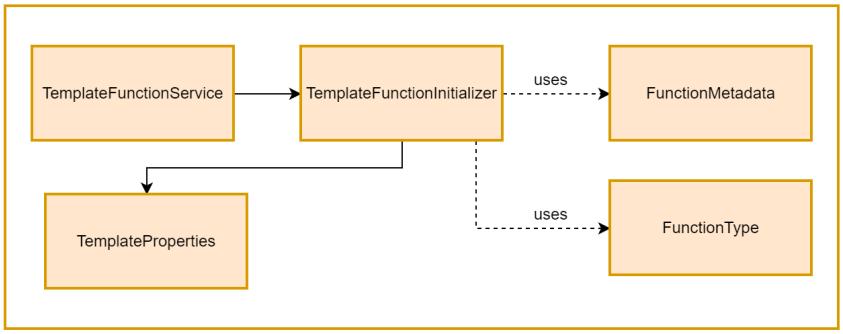


Figure 5.7: Specification of template component

ming syntax increases the level of abstraction and lowers an entry-level for constraint modeling.

5.5 Constraint validation

After explaining the domain layer of the specification and describing the essential services, it is time to look at the constraint creation and validation workflow. Figures 5.9-5.10 present two sequence diagrams indicating how the constraint framework can supplement an entity model with additional business rules. A detailed explanation with some remarks is followed below.

As specified, the sequence diagrams have three actor roles: client application, end-user, and constraint modeling framework. The process of constraint definition can be summarized by the subsequent steps:

1. An end-user opens a view with a model element that is to be enriched with a new constraint
2. A client-side software calls the rule engine to get all static constraint templates
3. A client application returns all the necessary information about a classifier and constraint templates
4. An end-user constructs the skeleton of a future constraint in the context of an opened model element by using delivered templates
5. An end-user fills the placeholders with meaningful data, thus conceptually finishing the constraint definition
6. An end-user submits a constraint. Further actions happen in the background.
7. A client-side application calls the framework to check the correct constraint definition
8. If the constraint does not violate the semantics of the domain schema, a consumer saves it

Subsequently, upon instantiating a model element, it must be ensured that all the constraints are valid in the context of a created instance. Hence, the following steps are relevant while checking constraints:

1. All the constraints associated with the model element an instance is instantiated from must be fetched

```
{
  "uuid": "fd33d40c-8c17-470f-8feb-d3161b5b11d2",
  "name": "Provide a meaningful name for a constraint.",
  "modelElementUuid": "#",
  "modelElementType": "SoftwareEngineer",
  "violationMessage": "Provide a meaningful message that will be delivered in
    case of constraint violation",
  "violationLevel": "ERROR",
  "constraintFunction": {
    "@type": "LOGICAL_FUNCTION",
    "name": "AND",
    "booleanFunctions": [
      {
        "@type": "RANGE_BASED_FUNCTION",
        "name": "GREATER_THAN",
        "attribute": "Provide a valid attribute in the context of the target
          model element. Example: <SoftwareEngineer>name.",
        "params": {
          "value": "Replace this placeholder with a concrete value according
            to the function description!"
        }
      },
      {
        "@type": "RANGE_BASED_FUNCTION",
        "name": "LESS_THAN",
        "attribute": "Provide a valid attribute in the context of the target
          model element. Example: <SoftwareEngineer>name.",
        "params": {
          "value": "Replace this placeholder with a concrete value according
            to the function description!"
        }
      }
    ]
  }
}
```


Figure 5.8: Template constraint definition

2. To validate constraints, the framework must be supplied with a subgraph of instances used as a context for evaluation. In order to prevent the loading of the entire graph into memory upon each constraint checking, the specification requires only those model elements that are part of a constraint definition. Hence, a client application sends a plain constraint to the rule engine
3. The framework determines the required model elements to be included in a graph and sends the response to the client
4. Based on the response, the client application constructs a subgraph as part of a constraint context
5. The created subgraph and a constraint are returned to the constraint modeling framework. If the subgraph does not correspond to the abstract syntax, it must first be mapped using an instance mapper
6. Upon receiving a graph and a constraint in an abstract syntax, the framework maps each of the acquired parameters to a platform-specific representation
7. The rule engine validates a constraint against a provided subgraph
8. After evaluation, the constraint dispatches a constraint validation report to the client application

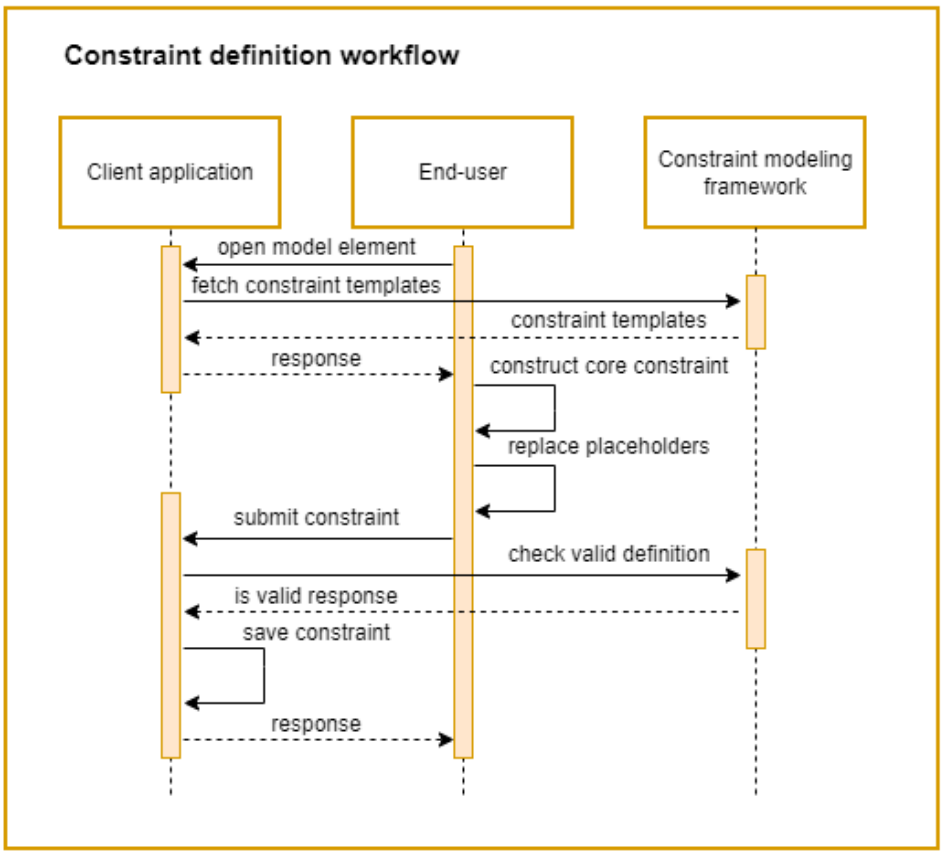


Figure 5.9: Constraint definition workflow

Moreover, the structure of a validation report returned to a client application is shown in figure 5.11 and has the following properties:

1. The identifier of an instance used as a context for evaluation
2. Name of the evaluated constraint
3. The name of a model element that a constraint was created for
4. Boolean flag to indicate whether a constraint is valid
5. Violation result that considers the level of violation and whether a constraint is valid. A client application can use the state of this variable to customize the workflow of a constraint violation handling
6. Violation message that informs about the cause of a faulty violation

In summary, this section has outlined how to integrate the constraint modeling framework into the workflow of client-side applications. By defining constraints, the framework internally enforces that every defined rule must adhere to the domain schema of the specification. Moreover, by validating rules, clients benefit from the constraint engine using minimal resources and providing sensible reports upon completion.

5.6 Synchronization mechanisms and their integration

Without providing additional integrity mechanisms, constraints that are applied to adaptive entities will eventually get stale, leading to inconsistencies or incorrect results. One of the pre-

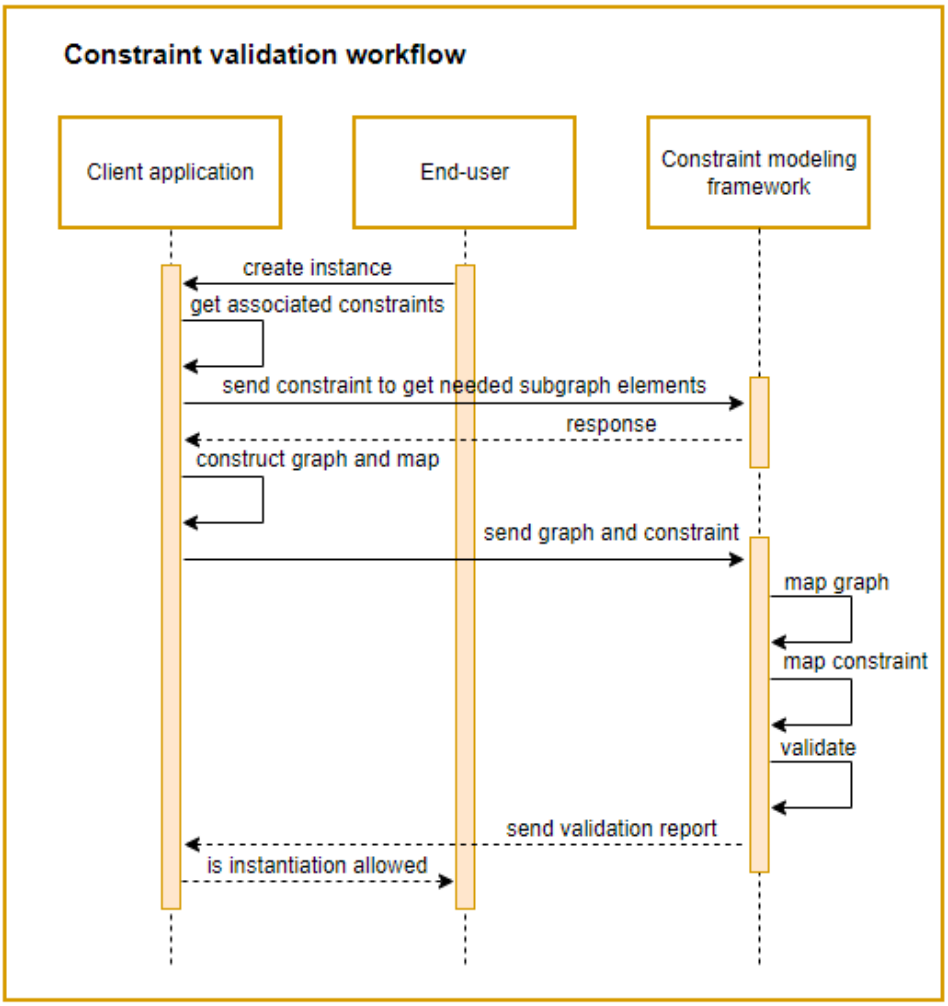


Figure 5.10: Constraint validation workflow

vious chapters focused on viable ways to overcome this issue. Back there, it was concluded that a straightforward approach to reevaluating all the constraints upon each model or graph space modification is not an option because it would lead to the utilization of tremendous computational resources, performance degradation, and, thus, breaking our system requirements. Instead, an alternative solution was suggested that would allow linking related entities or instances that associate with each other via constraints.

We classify two types of backward links depending on the integration level they provide. As such, there are model and instance backward links. Each of them serves a different purpose and intention. Model backward links are required to keep the definition of constraints up-to-date or in synchronization with the model space. If some part of a model schema changes, the respective constraints linked to the changed model elements must also be updated. Currently, the framework supports constraint synchronization in a semi-automatic fashion. Meaning that all updates of a model schema will lead to the automatic correction of affected constraints. Whereas deletion of some elements from the domain would require the intervention of an end-user. On the other hand, instance backward links are intended to keep the instance space adherent to constraints in the presence of dynamic linking operations. To be more specific, the change in one of the instances that are linked with a context instance must trigger the constraint evaluation of not only the updated instance but also check constraints applied to the context instance if constraints in the space of a context instance have references to the changed instance.

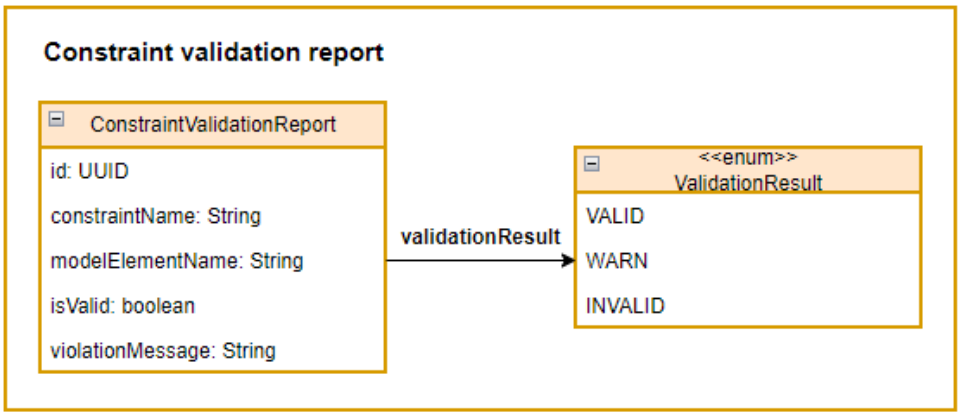


Figure 5.11: Constraint validation report

The domain space of integrity links is depicted in figure 5.12. Even though model and instance backward links are different in their intention of use, they are similar in structure. Both of them have a unique identifier, reference to a target model/instance element, and context model/instance element. One point to mention is that backward links must be integrated into the workflow of some operations on the side of a client application.

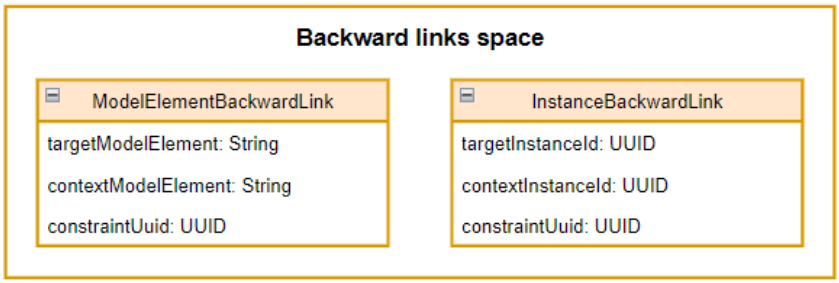


Figure 5.12: Domain space of backward links

5.6.1 Model backward links

Model backward links must be created upon saving constraints. They must be checked while updating or deleting some properties of a model element. Finally, they must be removed as soon as a constraint that is associated with them is removed. Figures X-Y explain how model backward links can be used to enforce the integrity of constraints in dynamic adaptive models. The additional steps that are needed to be included in the workflow to integrate backward links are highlighted in red. After synchronizing constraints, the constraint engine must return an integrity report. The class diagram of the report is shown in figure X and intends the following purpose. An integrity report is always created in the context of a modified model element. Consequently, depending on the type of operation, the integrity report must either include an old and a new property name or just the name of a property that was deleted from a schema definition. Finally, an integrity report must include a set of synchronized or invalid constraints depending on the action of an end-user.

Constraint definition with integrity mechanisms

Integrating model backward links upon constraint definition can be summarized by the following number of steps:

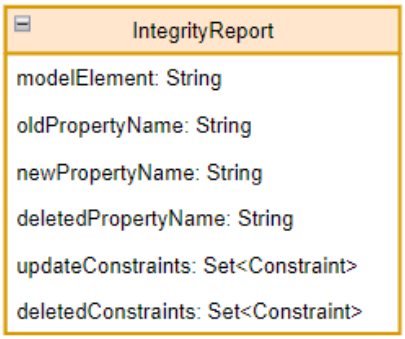


Figure 5.13: Integrity report

1. An end-user submits a newly created constraint
2. A client application calls the constraint framework to check the syntax
3. Upon getting a response, the client software sends the constraint to resolve backward links **(new)**
4. The framework retrieves all constraint functions from the constraint definition **(new)**
5. For every function that supports navigation, the framework retrieves model elements that a constraint refers to via navigation **(new)**
6. For each retrieved type, the framework creates a backward link **(new)**
7. A set of backward links is set to the client **(new)**
8. The client application is responsible to persist received links **(new)**
9. A constraint is saved
10. An end-user is notified about the status of the operation

Updating model schema with integrity mechanisms

The following actions must be applied to synchronize constraints upon changing the schema of a model element:

1. An end-user should edit some parts of an entity definition
2. An end-user must submit the element to be saved
3. The client application fetches all constraints defined in the context of the changed element
4. The client application fetches all constraints defined in the context of other model elements but existing in backward links **(new)**
5. All the constraints are submitted to the constraint framework to synchronize them **(new)**
6. The framework analyzes each constraint and finds those parts in the constraint definition that match the modified property **(new)**
7. Those parts of constraints are replaced with a new property value **(new)**

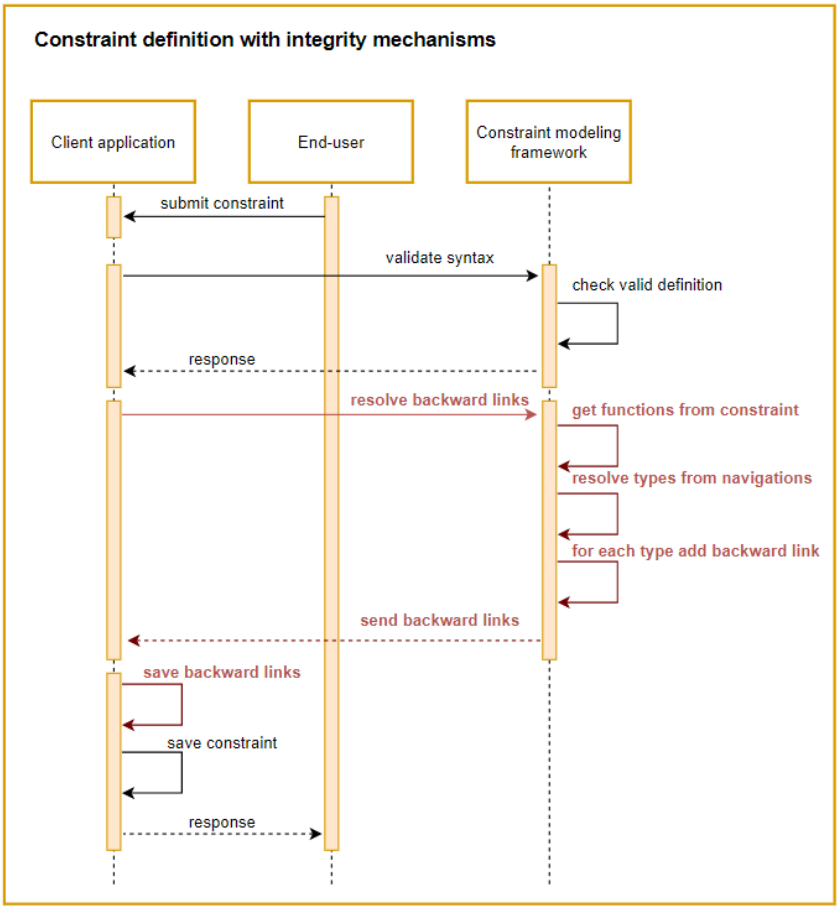


Figure 5.14: Constraint definition with integrity mechanisms

8. Based on the updated constraints, an integrity report is constructed **(new)**
9. The constraint framework sends the integrity report to a client application **(new)**
10. An end-user is provided with the integrity report and notified about the status of the operation **(new)**

Deleting parts of model schema with integrity mechanisms

If an end-user removes an attribute or an association of a model element, then before persisting changes, the following steps must be performed by a client application and the constraint framework.

1. An end-user should edit some parts of an entity definition
2. An end-user must submit the element to be saved
3. The client application fetches all constraints defined in the context of the changed element
4. The client application fetches all constraints defined in the context of other model elements but existing in backward links **(new)**
5. All the constraints are submitted to the constraint framework to get those that are invalid **(new)**

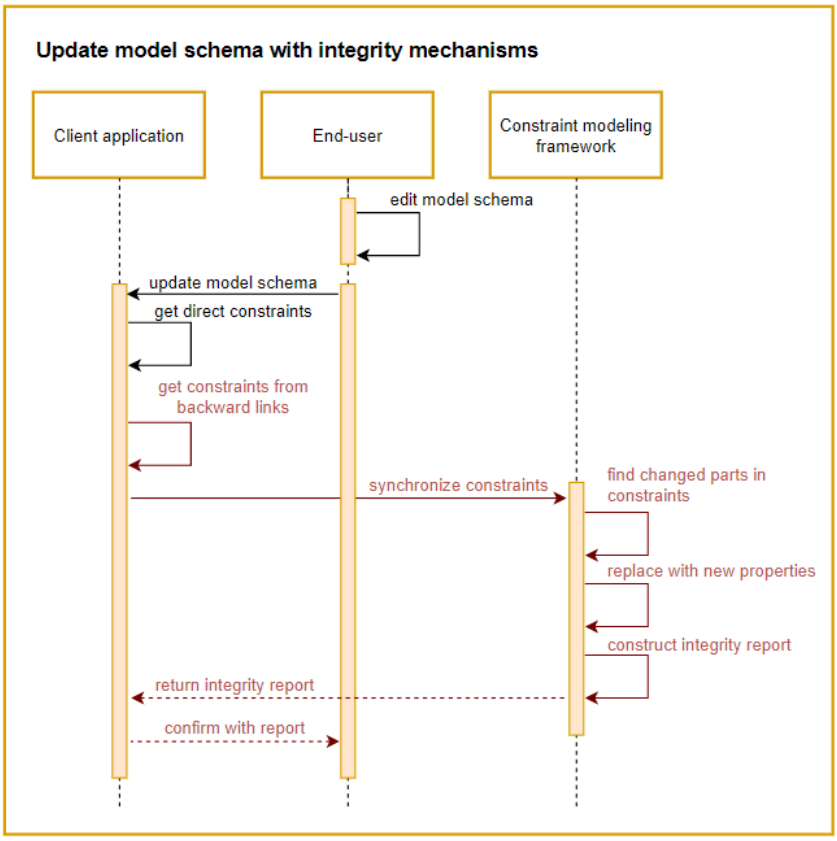


Figure 5.15: Updating model schema with integrity mechanisms

6. The framework analyzes each constraint and finds those constraints whose definition partially matches the deleted property **(new)**
7. Those constraints are added in the set of invalid constraints **(new)**
8. Based on the invalid constraints, an integrity report is constructed **(new)**
9. The constraint framework sends the integrity report to a client application **(new)**
10. An end-user provided with an integrity report must either remove invalid constraints or correct them **(new)**
11. Consequently, an end-user submits all the changes **(new)**

5.6.2 Instance backward links

An instance backward link must be created if a constraint is restricting several model elements (e.g., via a simple association or instance-object-nets constraint), and this instance is linked with another instance whose type is mentioned in the constraint definition. Hence, upon updating a linked instance, all of its constraints as well as all context constraints will be evaluated, thus ensuring the consistency of constraint enforcement. If a link between two instances or a constraint is removed, the respective instance links must also be removed.

Adding instance backward links

The process of linking two instances with enabled integrity mechanisms could be described in the following steps.

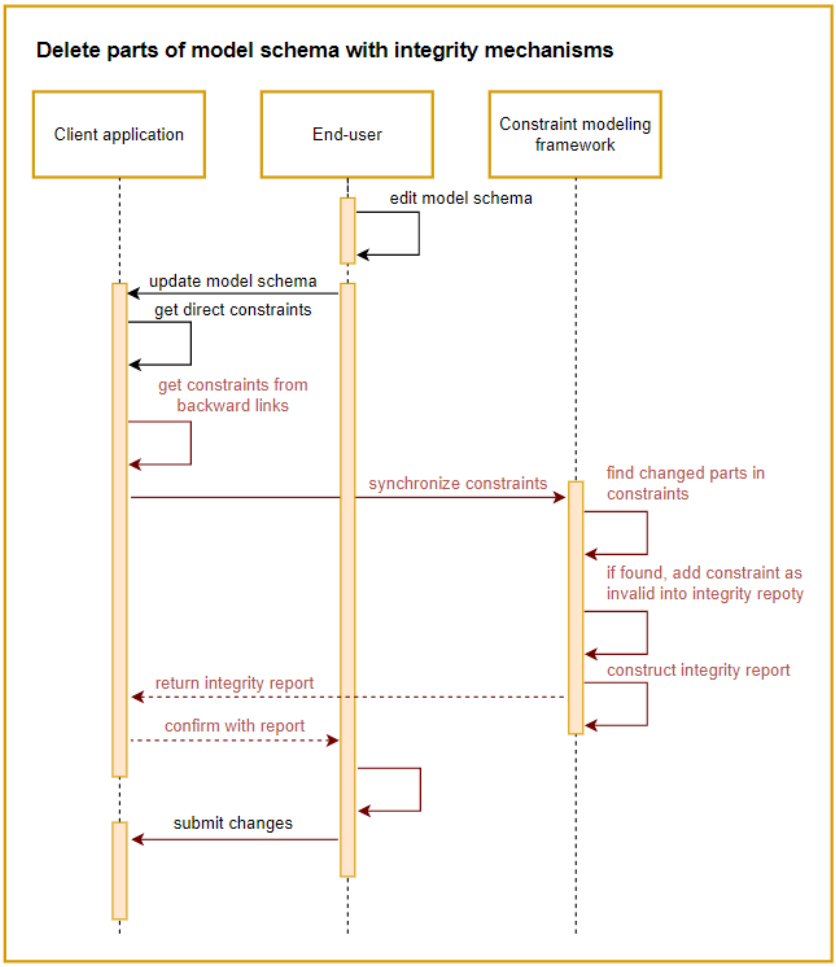


Figure 5.16: Deleting parts of model schema with integrity mechanisms

1. An end-user either instantiates a model element or selects an already created instance that is to be updated
2. An end-user links the selected instance with another instance by a link
3. An end-user initiates a call to save changes
4. The client application must fetch all the constraints related to the context instance **(new)**
5. The client application must retrieve the link that was just added by an end-user **(new)**
6. The client application sends a request to the constraint framework to resolve backward instance links based on the added link and the set of fetched constraints **(new)**
7. The framework checks every constraint whether it has a reference in its definition to the linked instance **(new)**
8. If it is the case, the constraint framework creates an instance backward link **(new)**
9. All created backward links are returned to the client application **(new)**
10. The client application saves backward links **(new)**
11. Finally, the client software must reply to an end-user whether their request to save changes was accepted or aborted

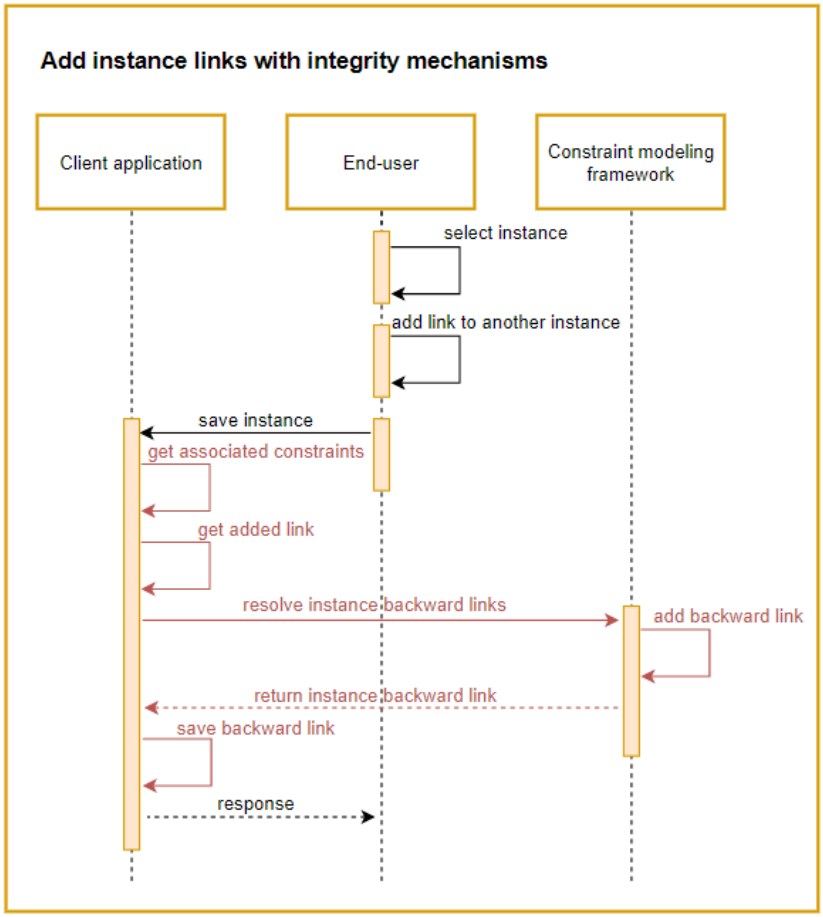


Figure 5.17: Linking instances with integrity mechanisms

Updating linked instances ensuring integrity

Whenever a linked instance is updated by an end-user, it must be ensured that all relevant constraints are not violated as a result of the operation. Hence, the following workflow could be used to achieve the desired integrity.

1. An end-user selects an instance that is already linked by one of the backward links
2. An end-user updates an instance
3. A request to save an instance is received by a client application
4. A client software fetches all directed constraints associated with an instance
5. A client software fetches instance backward links associated with a given instance
6. A consumer sends a request to the framework to validate constraints
7. Direct constraints, as well as the ones resolved from links, are evaluated by the rule engine
8. A constraint validation report is sent to the client application
9. The client application notifies an end-user about the status of a requested operation

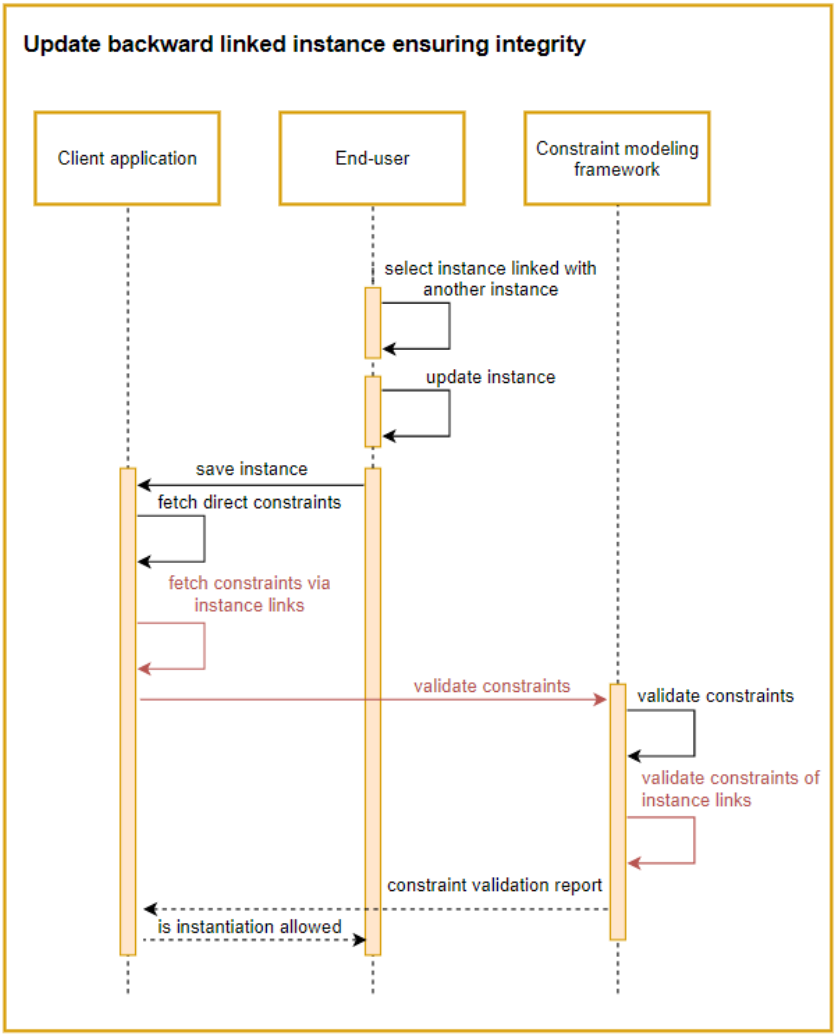


Figure 5.18: Updating a backward linked instance ensuring integrity

5.7 Summary

This chapter focused on the architectural structure, design, and behavioral aspects of specification, particularly on the domain layer of the specification that shapes the process of constraint definition and validation. It also covered the core services that make up the conceptual logic of a constraint modeling framework. Moreover, this chapter offered a reader a number of sequence diagrams that illustrate how a concrete implementation of the specification could be integrated into existing software systems. Additionally, the chapter delves into the topic of guaranteeing constraint integrity in the presence of dynamic model changes. The comprehensive explanation of all architectural aspects and design decisions form a robust foundation to implement the specification completely through Gremlin and SHACL as rule engines.

6 Implementation

Encapsulating the domain layer and the structure of a constraint modeling framework into a separate module produces a clear roadmap of how and what should be done during the implementation phase. The specification proposed in the previous chapter reflects the requirements, functionality, and fundamental behavior of the framework. In this chapter, we combine the results of related works with our specification module to implement an end-user constraint modeling framework using SHACL and Gremlin technologies. As a result, every implementation adheres to the outlined specification reducing development time and error probability.

6.1 General overview

As explained in the previous chapter, the specification exposes a set of mappers and services. Some of the services are implemented by specification by default or can optionally be overwritten by clients. Another set of abstract services must be implemented by a platform-specific rule engine. This category of classes does not have a default implementation because the specification is platform-agnostic and is not aware of any particular tool. Figure 6.1 illustrates the components relevant to validating constraints and shows the connection between the abstract and platform-specific elements.

Since the purpose of every abstract service and mapper was explained in the previous section, we refrain from the repetitive detailed description of the class diagram. One thing to observe is that every concrete implementation must have its own domain layer that should comprise two entities: *InstanceGraph* and *Constraint*. Both of them come into play during the conversion of an abstract instance graph and constraints into their environment-specific representations. Subsequently, a mapped graph and constraint are passed further to a constraint checker for further evaluation. Apart from it, every constraint modeling implementation must have a domain class *FunctionRegistry*. A function registry stores links between the name of a function and its concrete implementation on a particular platform. This class is then used by *FunctionMapper* to map an abstract constraint.

6.2 Gremlin domain

Out-of-the-box Gremlin does not have a definition of constraints. Hence, it cannot instantly be used as a constraint language. Being a system designed for traversing graphs, Gremlin operates in the context of edges and vertices, providing mechanisms to explore graph-like structural data. To adjust Gremlin to constrain adaptive entities, it is necessary to create a custom DSL.

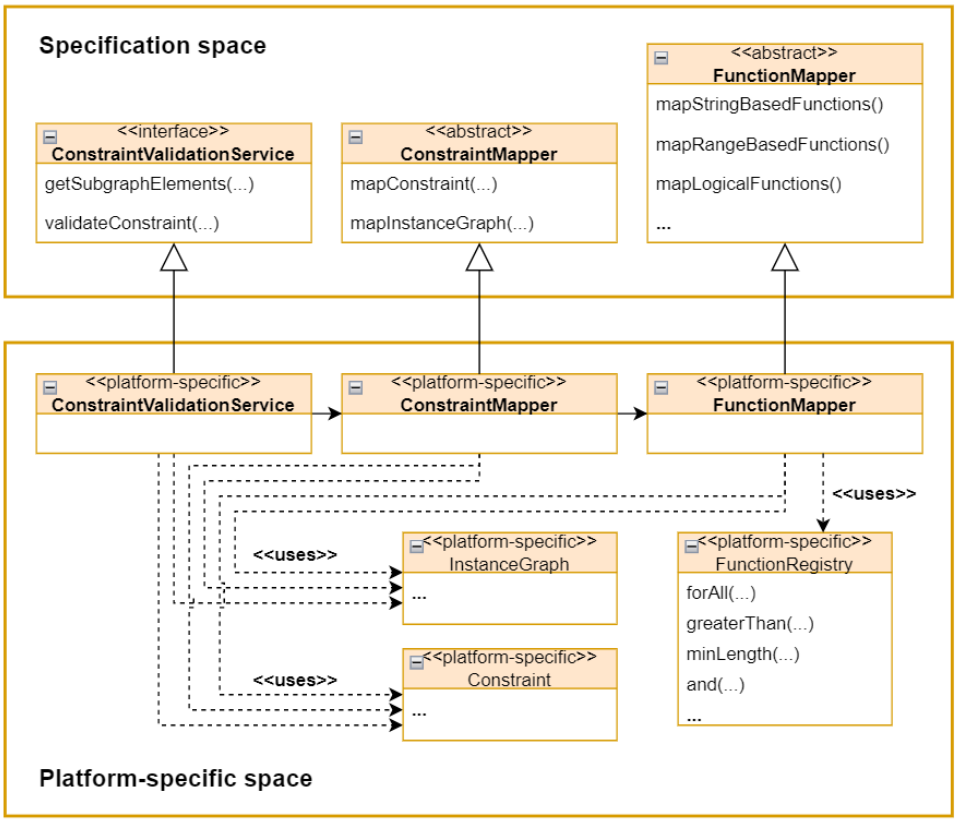


Figure 6.1: Implementing specification diagram

The domain layer for the constraint modeling framework is based on the extension of two Gremlin classes:

1. *GraphTraversal* is a key interface component of the whole Gremlin implementation. It provides a set of functions and utilities to manipulate and query graphs. The operations can be chained with each other allowing the definition of complex queries.

2. *GraphTraversalSource* is an interface to create an instance of a traversal. Every traversal encapsulates a graph with concrete data consisting of vertices and edges. This interface is essential and serves as a context for executing Gremlin queries

To draw a comparison with the domain of the specification, one could argue that *GraphTraversal* is well-suited to define constraints available for end-users, whereas *GraphTraversalSource* can be utilized as a platform-specific implementation of an abstract graph syntax.

Moreover, to facilitate Gremling supporting constraints, we use the *@GremlinDsl* annotation. *@GremlinDsl* allows specifying domain-specific constructs for graph traversals. It is used to mark Java methods that should be included in the DSL, allowing the definition of custom traversal steps. Every custom method should return a *GraphTraversal* according to the documentation [Tin21a].

Hence, by extending *GraphTraversal* and applying the *@GremlinDsl* annotation, we create a domain-specific class that encapsulates static constraints. Figure 6.4 illustrates an example of a constraint definition with Gremlin. The constraint is applied in the context of domain-specific *GraphTraversalSource* and accepts the name of an attribute and the minimum value it should have to pass the constraint evaluation. The traversal strategy fetches the required attribute from a context instance and applies trivial comparison and arithmetic operations. As a result, the method returns a boolean value wrapped into a traversal where true means that a

constraint is valid, whereas false indicates the opposite. All other constraints are implemented in a similar fashion.

```
default GraphTraversal<S, Boolean> greaterThan(String attribute, String value) {
    return values(attribute).map(traverser -> {
        long elementValue = Long.parseLong(String.valueOf(traverser.get()));
        boolean valid = elementValue > Long.parseLong(value);
        LOGGER.info("GreaterThan({}({}), {}) -> valid: {}", attribute, elementValue, value, valid);
        return valid;
    });
}
```

Figure 6.2: Range-based constraint with Gremlin DSL

Operating on the custom implementation based on *GraphTraversalSource*, we hide the graph-related attributes like edges and vertices and expose users only terms from the domain of constraint modeling. For example, as shown in figure 6.3, the platform-specific implementation has methods to add an instance into a graph, check whether a given instance is part of a graph, or link two instances.

```
/**
 * Adds instance to the Gremlin graph.
 *
 * @param instanceElement See {@link InstanceElement}
 */
2 usages  Anton Skripin
public void addInstance(InstanceElement instanceElement) {
    var instanceVertex : GraphTraversal<Vertex, Vertex> = this
            .addV(instanceElement.getInstanceOf())
            .property( key: "uuid", instanceElement.getUuid());

    instanceElement.getSlots().forEach(slot -> {
        instanceVertex.property(slot.getKey(), slot.getValue());
    });
    instanceVertex.iterate();
}

/**
 * Links two instances in a graph by link.
 *
 * @param link See {@link Link}
 */
2 usages  Anton Skripin
public void linkTwoInstances(Link link) {
    if (doesInstanceExists(link.getTargetInstanceUuid())) {
        this.addE(link.getName())
                .from(this.V().has(UUID_PROPERTY, link.getInstanceUuid()).next())
                .to(this.V().has(UUID_PROPERTY, link.getTargetInstanceUuid()).next())
                .iterate();
    }
}

1 usage  Anton Skripin
private boolean doesInstanceExists(String uuid) { return this.V().has(UUID_PROPERTY, uuid).hasNext(); }
```

Figure 6.3: Domain-specific traversal source

6.3 Shacl domain

SHACL is a web standard that provides a vocabulary and syntax for describing constraints on RDF data models. It allows the specification of rules and conditions that a graph must conform to. The core class to create a domain-specific domain layer is *ModelCom*. *ModelCom*

is an implementation of the *Model* interface that represents an RDF model. It provides a set of methods for manipulating and querying RDF data, including adding and removing statements, iterating over the nodes in the model, and querying specific patterns or properties [The23].

According to [The23], *ModelCom* is designed to be a high-performance implementation of the *Model* interface, optimized for large-scale RDF data processing. It uses a variety of techniques to improve performance and reduce memory overhead, such as caching frequently accessed data and optimizing queries to minimize the number of triples that need to be examined during query processing. As a result, two entities, namely *ShaclConstraintShape* and *ShaclInstanceGraph*, were created as an extension of the *ModelCom* class.

To define user-driven constraints, *ShaclConstraintShape* is used as a container. An example of a constraint schema using Shacl is presented in figure 6.4. A constraint is evaluated in the context of *ShaclInstanceGraph* that serves as a platform-specific implementation for an abstract graph definition. In particular, this constraint accepts an attribute and a minimum value it must hold. The *path* property is used to specify the route to a property that is being constrained. The *minExclusive* property declares that a given value must be greater than a specified threshold. The return value of the constraint definition is *Resource* which is submitted to the Shacl rule engine for further evaluation. All the other constraints are implemented in an identical way by applying Shacl properties to define the shape of a context instance and returning *Resource*.

```
public Resource greaterThan(String attribute, String value, boolean nested) {
    Resource resource = getResource(nested);
    return resource.addProperty(SH.property, this.createResource()
            .addProperty(SH.path, this.createProperty( uri: CONSTRAINT_NAMESPACE + attribute))
            .addProperty(SH.minExclusive, value, getXsdDatatypeByValue(value)));
}
```

Figure 6.4: Range-based constraint with Shacl DSL

Moreover, *ShaclInstanceGraph* is needed to provide domain-specific methods to manipulate graphs in the context of constraint modeling. In a similar way to Gremlin, this class exposes methods to manage instances in a platform-specific graph implementation.

6.4 Constraint mapper

Mapping abstract elements to their platform-specific implementation is an essential step before evaluation. An environment-specific constraint mapper consists of two methods: graph and constraint mappers. A constraint checker must be provided with the context for evaluation, in our case, with a graph. Hence, one of the methods maps an abstract graph to its platform-specific representation. The process of mapping is shown in figure 6.5. It is worth noting that since every implementation follows the same steps outlined in the specification, the process of graph transformation looks the same way both for Gremlin and Shacl implementations. The mapping is concise and intuitive because all technology-related functions are encapsulated, exposing only domain-specific operations.

Moreover, an abstract constraint that is created by an end-user must also be mapped to its concrete implementation. As outlined before, every constraint contains a root function. A root function accepts nesting functions, thus allowing the definition of complex constraints. As a result, the structure of a final function constraint is composite and is resolved recursively. The transformation of a function starts from its leaf (very bottom) and finishes at its root. A platform-specific constraint must have a starting context from where a rule engine can begin the evaluation. The evaluation context is resolved dynamically and depends on a platform-specific graph. The mapping between an abstract and platform-specific constraint is shown in figure 6.6. The algorithm of mapping is outlined in figure 6.7. However, to map the name of

```
3 usages  Anton Skripin
@Override
public ShaclConstraintData mapToPlatformSpecificGraph(List<InstanceElement> graph) {
    LOGGER.info("Total number of graph elements to be mapped: {}", graph.size());
    ShaclConstraintData model = new ShaclConstraintData();
    measureExecutionTime( name: "create elements", () -> graph.forEach(model::createInstance));
    measureExecutionTime( name: "create links", () -> graph
            .stream()  Stream<InstanceElement>
            .filter(instanceElement -> Objects.nonNull(instanceElement.getLinks()))
            .flatMap(instanceElement -> instanceElement.getLinks().stream())  Stream<Link>
            .forEach(model::addLinkToInstance));
    return model;
}
```

Figure 6.5: Mapping an abstract graph to Shacl

a function with its concrete implementation, we require one more service, namely a function mapper. An explanation of it follows in the next section.

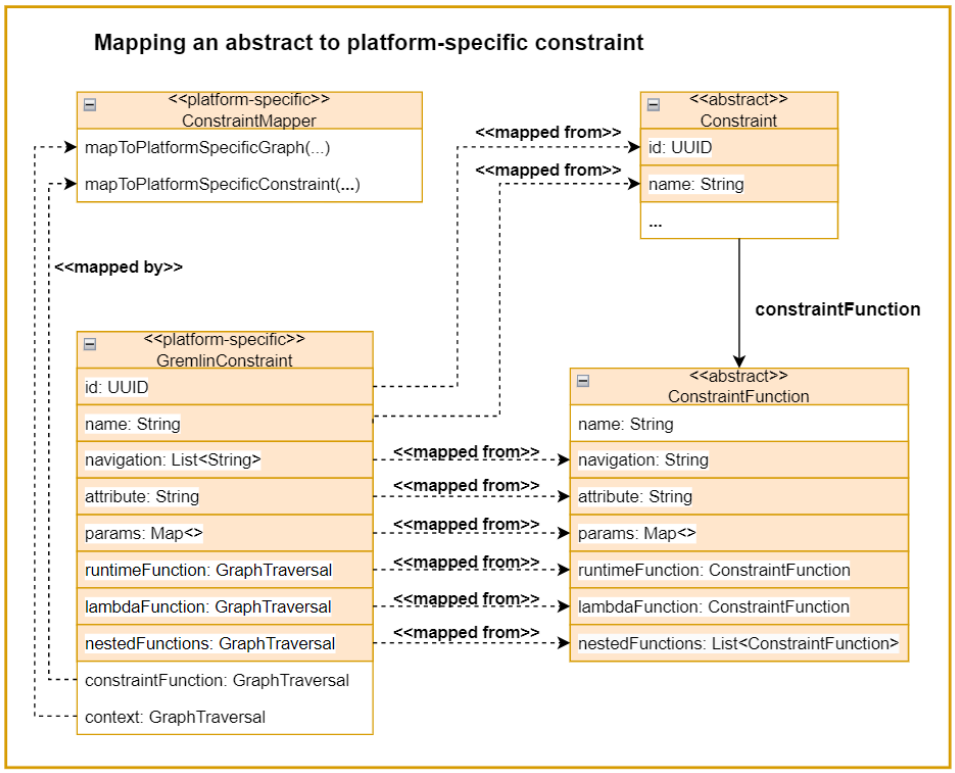


Figure 6.6: Mapping an abstract to platform-specific constraint. Diagram.

6.5 Function mapper

An abstract constraint consists of a set of functions. Every abstract function that is defined by an end-user contains its type, name, and other attributes. All this should be mapped to a platform-specific technical space. The function mapper is designed to provide a convenient way to map abstract constraint functions to their concrete implementations at runtime. This is achieved through the use of a map, where the key is the name of the abstract function, and the value is a platform-specific implementation. The purpose of such a design is to enable the platform-specific constraint framework to receive a constraint consisting of abstract func-

```
function.params().ifPresent(gremlinConstraint::setParams);
function.runtimeFunction().ifPresent(gremlinConstraint::setRuntimeFunction);
function.attribute().map(AttributeUtils::getAttributeRoot).ifPresent(gremlinConstraint::setAttribute);
function.navigation().map(NavigationUtils::getNavigationRoot).ifPresent(gremlinConstraint::setNavigation);

function.lambdaFunction() Optional<ConstraintFunction>
        .map(lambdaFunction -> mapFunction(uuid, lambdaFunction)) Optional<GraphTraversal<capture of ?, Boolean>>
        .ifPresent(gremlinConstraint::setLambdaFunction);
function.booleanFunctions()
        .ifPresent(booleanFunctions -> booleanFunctions
                .forEach(booleanFunction -> gremlinConstraint.addNestedFunction(mapFunction(uuid, booleanFunction))));
```

Figure 6.7: Mapping an abstract to platform-specific constraint. Implementation.

tions and resolve them dynamically by their name. This is a powerful feature that allows the implementation to be flexible and adaptable to changing requirements.

To showcase the process of functional mapping, figure 6.8 provides a small snippet that transforms one of the range-based functions. This particular *lessThan()* function is designed to take a platform-specific constraint that contains attribute name and value. All this information is available as a result of mapping a constraint that occurs beforehand. To link the schema of a constraint function with concrete values provided by an end-user, a function mapper retrieves the metadata about every required constraint on demand from the specification. Since the core shape of a function mapper is defined in the specification, all platform-specific implementations share the same structure and logic for converting functions from their abstract representation to an environment-specific one.

```
public class GremlinFunctionMapper extends AbstractFunctionMapper<GremlinConstraint, GraphTraversal<?, Boolean>> {

    Anton Skripin *
    @Override
    public Map<String, Function<GremlinConstraint, GraphTraversal<?, Boolean>>> mapRangeBasedFunctions() {
        Map<String, Function<GremlinConstraint, GraphTraversal<?, Boolean>>> mapper = new HashMap<>();

        mapper.put(GREATER_THAN, gremlinConstraint -> {...});
        mapper.put(GREATER_THAN_OR_EQUALS, gremlinConstraint -> {...});
        mapper.put(LESS_THAN, gremlinConstraint -> {
            String attribute = gremlinConstraint
                    .attribute()
                    .orElseThrow(() -> new GraphConstraintException("LessThan() must have an attribute!"));
            String value = gremlinConstraint
                    .params() Optional<Map<String, String>>
                    .orElseThrow(() -> new GraphConstraintException("LessThan() must have a value attribute")) Map<
                    .get(FUNCTION_TO_PARAMETER_NAMES.get(LESS_THAN).stream().findFirst().get());
            return gremlinConstraint.context().isPresent() ?
                    gremlinConstraint.context().get().lessThan(attribute, value) :
                    __.lessThan(attribute, value);
        });
        mapper.put(LESS_THAN_OR_EQUALS, gremlinConstraint -> {...});
        mapper.put(EQUALS, gremlinConstraint -> {...});
        return mapper;
    }

    new *
    @Override
    public Map<String, Function<GremlinConstraint, GraphTraversal<?, Boolean>>> mapCollectionBasedFunctions() {...}
```

Figure 6.8: Resolving abstract functions to their Gremlin implementations

By using a map to store the concrete implementations, the function mapper provides a scalable and extensible solution that can easily accommodate new constraints and functions as needed.

6.6 Validation service

The validation service abstracts all the complexities related to mapping domain entities. It also serves as a gateway entry for client applications. In general, this function must be provided with three attributes: an identifier of an instance that will be used as a starting point for evaluation, an abstract subgraph that serves as a context, and a constraint itself. The process of constraint validation is depicted in figure 6.9 and can be summarized in the following steps:

```
@Override
public ConstraintValidationReport validateConstraint(String uuid, List<InstanceElement> subgraphForValidation, Constraint constraint) {
    try (
            ConstraintGraphTraversalSource gremlinGraph = constraintMapper.mapToPlatformSpecificGraph(subgraphForValidation);
            GraphTraversal<?, Boolean> gremlinConstraint = constraintMapper.mapToPlatformSpecificConstraint(uuid, constraint)) {
        boolean isValid = gremlinGraph.isValid(gremlinConstraint);
        gremlinConstraint.close();
        gremlinConstraint.notifyClose();
        gremlinGraph.close();
        return new ConstraintValidationReport(
                uuid,
                constraint.getName(),
                constraint.getModelElementType(),
                isValid,
                constraint.getViolationLevel(),
                constraint.getViolationMessage());
    } catch (Exception e) {
        throw new ConstraintValidationException(e);
    }
}
```

Figure 6.9: Platform-specific validation service

1. Using a constraint mapper, an abstract graph is transformed into a platform-specific one
2. Using a constraint mapper, an abstract constraint is converted into a platform-specific one
3. A constraint checker takes as an input an environment-specific constraint and graph
4. Based on the result of the evaluation, a constraint validation report is generated and returned to the client application

6.7 Summary

This chapter thoroughly explained the principles involved in implementing a constraint modeling framework. It outlined the key components, including the domain layer, that should be part of a platform-specific implementation. The workflow for constraint validation and the functioning of involved components is also described in detail. The environment-specific implementation with Shacl and Gremlin both adhere to the well-defined constraint modeling specification that highlights one more time the simplicity and similarity of integrating new tools to develop the abstract specification. In short, this section outlined the key principles involved in implementing a constraint modeling framework. Hence, it might serve as a valuable resource for anyone seeking to either implement the constraint modeling specification or develop their own framework with an identical use-case.

7 Evaluation

7.1 Goal

The previous chapter explained how the constraint modeling specification for entity models could be integrated into the workflow of user-driven applications and implemented by concrete tools. Specifically, we focused on developing using Gremlin and Shacl. While both implementations are capable of defining and checking constraints at runtime, they have distinct strengths and weaknesses in terms of expressiveness, performance, and other factors. In this chapter, we evaluate both implementations based on the formulated requirements in one of the earlier chapters. The primary goal is to provide insights into their suitability and potential trade-offs for different use cases.

7.2 Evaluation criteria

Based on the requirements defined for a user-driven constraint modeling framework, we can highlight some prerequisites that relate directly to the tool used for constraint validation. Therefore, the following points are selected as primary criteria for evaluation based on the stated system specification:

- Expressiveness and support of different constraint types - the expressiveness of a constraint language is a crucial characteristic of applying business requirements to adaptive models. Therefore, it is imperative to assess what specified constraints a particular tool is able to express
- Support for extendability - apart from the pre-existing constraint functions available during compile time, a specific tool must allow the designing of new types of constraints during runtime
- Performance refers to the ability of a system to process constraints and provide results within an acceptable timeframe. During the evaluation, we consider response and processing time as crucial factors in determining how well an implementation with each concrete technology can handle complex constraints on a large number of entities

Apart from it, we add additional assessment measures to compare Gremlin and Shacl in the context of constraint modeling. Including them would provide a more comprehensive overview of a concrete tool acting as the fundamental component to implement the abstract specification.

- Maintainability emphasizes whether software can be smoothly modified or extended over time to keep it operating as expected. This criterion takes into account the size of a technology community, its documentation, and the frequency of introducing breaking changes during software release. These factors collectively determine how efficiently developers can maintain and evolve a system within the software development lifecycle
- Interoperability describes the ability of a system to work seamlessly with other software, languages, or platforms. In summary, the meaning of this property might be acquired by the following questions. How well does a tool integrate with other technologies and systems? Can it efficiently work with different data formats, APIs, and programming languages?

7.3 Comparison

7.3.1 Expresiveness

While implementing the specification, it was discovered that Gremlin is capable of describing more comprehensive and non-trivial types of constraints than Shacl. The prototypical version of the framework targeted the development of 19 distinct constraint functions, thus covering 7 constraint types. A detailed explanation is provided below for each function constraint that has not been implemented in Shacl, along with the reasons for it.

1. Unique()
   - ***Definition*** - It enforces a particular attribute value to be unique across all instances of a specific model element.
   - ***Explanation*** - According to the documentation of Shacl [KK17], there is a *sh:uniqueLang* constraint. Enforcing this rule ensures that no two literals in a given property have the same language tag. However, this type of constraint is too specific and narrows down to checking only the subpart of the whole property. Therefore, extending this constraint to make it more generic to be applied to the entire property value is required to implement the Unique() function with Shacl.
2. FoAll()
   - ***Definition*** - It ensures that a specified condition holds true for all instance elements after navigation starting from a context instance.
   - ***Explanation*** - This function is impossible to implement in Shacl because it explicitly requires a minimum and a maximum number of matches of a particular property that must be present for a given resource to consider a submitted constraint valid. However, when evaluating this kind of constraint, the number of occurrences of the property depends on the number of instances resulting from the navigation operation, which cannot be known in advance.
3. IfThen()
   - ***Definition*** - It allows the creation of an optional constraint function that will be evaluated if and only if a conditional constraint holds.
   - ***Explanation*** - This function constraint is impossible to implement in Shacl simply because no equivalent or similar method is provided by its syntax. Moreover, Shacl constraints are primarily declarative, meaning they specify rules on data without providing operations or actions to perform on the input graph. As a result, it is limited in its ability to express conditional statements.

```
{
  "constraintFunction": {
    "@type": "RUNTIME_FUNCTION",
    "name": "An employee must not be older that 65 years old",
    "runtimeFunction": "values('age').is(P.lt('65'))"
  }
}
```

(a) Dynamic function with Gremlin

```
{
  "constraintFunction": {
    "@type": "RUNTIME_FUNCTION",
    "name": "An employee must not be older that 65 years old",
    "runtimeFunction": "sh:property [ sh:maxInclusive \"65\"^^<http://www.w3.org
      /2001/XMLSchema#int>; sh:path eumcf:age];"
  }
}
```

(b) Dynamic function with Shacl

Figure 7.1: Constraint type definition at runtime

4. IfThenElse()
   - ***Definition*** - It allows the creation of an optional constraint function that will be evaluated if and only if a conditional constraint holds. In addition, an alternative constraint function is supplied for evaluation if the initial condition is false.
   - ***Explanation*** - Since this function is the extended version of the *IfThen()* function and has conditional implications, it cannot be implemented in Shacl for the same reasons stated above.

7.3.2 Extendability

Both Shacl- and Gremlin-based implementations support the definition of new constraints at runtime. The capability to create new constraint types on-the-fly correlates with the possibility of concrete technology to produce and parse a constraint from plain text. An end-user defines a constraint as a string adhering to the syntax of a platform-specific tool. Upon submission, it will be dynamically added to the space of constraint types along with statically defined ones.

To create a new rule function at runtime with Shacl, we need *ModelCom* and *Shapes* classes. First, the service parses string into a resource presented in the Turtle format. Then, the *Shapes* utility class parses the created resource into a Shacl shape. Finally, the rule engine accepts and validates the shape as an input parameter.

Declaring a new constraint type by an an-end user takes only a small amount of effort for both implementations. As such, a user must only enter a query that should resolve into true or false. However, as shown in figure 7.1, the definition of a new function with Shacl might be more challenging to understand since it uses RDF syntax. In comparison, the syntax of Gremlin is more concise and resembles a natural language that seems more coherent.

7.3.3 Performance

Validating constraint entitles several steps. First, a provided graph must be transformed into its platform-specific representation. Then, an abstract constraint must be mapped to its platform-dependent form. Finally, a constraint is submitted to a rule engine for evaluation.

Each of these steps requires time and resources. Therefore, to reason which implementation is better in terms of performance, we should address each stage independently as well as in conjunction.

To estimate the performance capabilities of each platform-specific implementation, we developed a case study application that includes five experimental runs in total. The overview of every experiment is provided below:

1. Graph transformation experiment. An abstract graph is generated with a given number of instance elements related to each other. The experiment measures the time of mapping an abstract form to its platform-specific representation
2. Constraint mapping experiment. The data generation service spawns a constraint with a given number of nested attribute-based functions. The experiment measures the time of mapping an abstract constraint into its platform-specific format
3. Instance-object-nets constraint mapping experiment. A constraint with a provided number of nested functions with navigation to other instance elements is assembled. The experiment measures the time of mapping an abstract complex constraint into its environment-targeted entity
4. Constraint validation time experiment. This experiment omits mapping steps and focuses on measuring the needed time for a constraint engine to check a constraint with an arbitrary number of nested functions
5. Overall constraint validation experiment. This experiment combines all the aforementioned experiments into one overall procedure. As such, the service only generates an abstract graph and constraint. The rest of the workflow is tracked by time and reported accordingly

All of the experiments have been performed on a machine with the following technical specification. ***Processor***: Intel Core i7-8550U, CPU @ 1.80GHz 1.99 GHz. ***RAM***: 8.00GB (7.89 GB usable)

Graph transformation experiment

The experiment begins with an empty graph that gradually expands in size. In the final iteration, the graph comprises 500 instance elements linked through association links. The size of the graph is increased by 5 instance elements in each step. In total, the experiment was repeated 6 times. The initial execution served as a warm-up phase to reduce the impact of the JVM startup on the final results. The subsequent five executions are combined, and an average value is computed for each incremental step to eliminate any execution inaccuracies. Figure 7.2 depicts the performance graph, with the Gremlin's performance displayed in red and Shacl's in blue.

It can be observed that Shacl transforms graphs significantly faster than Gremlin. Once the number of instances in a graph exceeds fifty elements, the performance of Gremlin begins to lag behind. At the last iteration step, Shacl outperforms Gremlin by eight times. Such considerable differences in performance may be due to the different approaches used by the two tools to construct graphs. Shacl appends new instance elements directly as an RDF resource to the final graph, while Gremlin must first check if an instance already exists and then execute traversal steps on each operation. As a result, Shacl's approach may be more efficient than Gremlin's in terms of graph construction speed.

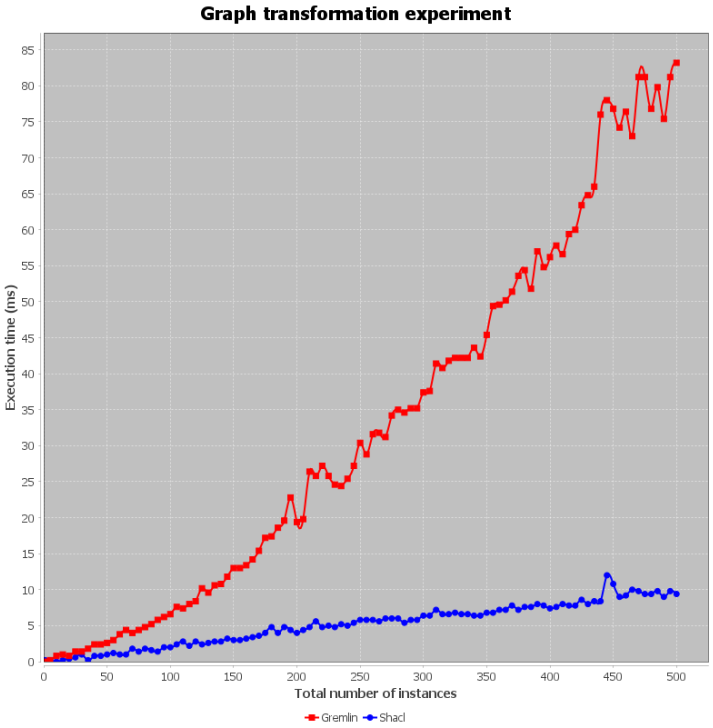


Figure 7.2: Graph transformation

Constraint mapping experiments

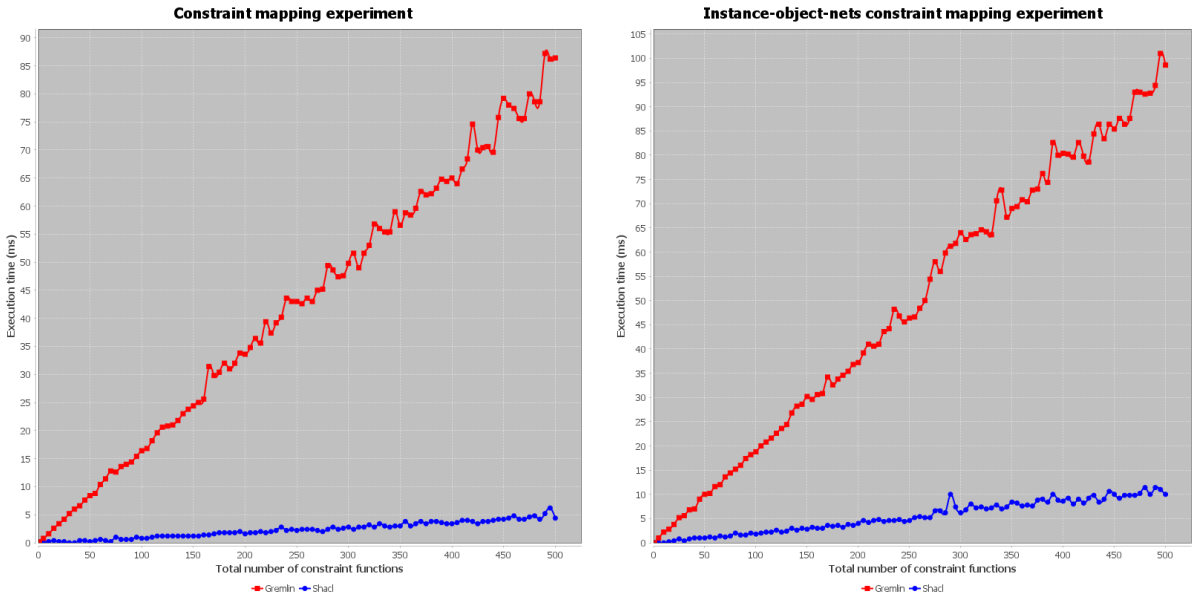
Both experiments begin with an empty set of initial constraint functions. The number of nested functions included in each constraint gradually increases by five functions per step. In the end, the number of functions in one constraint reaches five hundred. Figure 7.3a displays the mapping time for a simple constraint limited to only one instance element. In contrast, Figure **??** shows the performance of transforming a constraint where its functions span multiple instance elements. In both cases, Shacl significantly outperforms Gremlin. This significant performance difference can be attributed to the same factor as in the previous experiment: Shacl shapes are constructed by adding parts of a resource directly to a final shape, whereas Gremlin must apply traversal steps to a graph for each received constraint function.

Constraint validation time experiment

At the start of an experiment, the data generator creates an empty graph and constraint. The number of elements in both the graph and constraint functions increases by 200 with each incremental step. Following the mapping phase, a platform-specific graph and constraint are submitted to a rule engine. The experiment measures only the timeframe of the Gremlin and Shacl constraint engines needed to evaluate rules. Figure 7.4 demonstrates that the performance of both tools is fast and stays nearly identical regardless of input complexity.

Overall constraint validation experiment

The experiment addresses the constraint evaluation workflow starting from the first step of mapping abstract input parameters. This experiment summarizes all the previous runs into one workflow to demonstrate the efficiency of each tool. As expected, the results shown in figure 7.5 confirm one more time that Shacl surpasses Gremlim in terms of productivity. It is worth noting that the bottleneck of Gremlin is the initial construction of a traversal source and



(a) Mapping attribute-based constraints

(b) Mapping complex constraints

Figure 7.3: Constraint mapping time

traversal steps. In contrast, the engine itself can validate queries in a fast manner. In summary, under similar conditions and operational procedures, Shacl outperforms Gremlin on average by a factor of eight.

7.3.4 Mantainability

Both Shacl and Gremlin have their advantages and disadvantages when it comes to maintainability. Therefore, selecting a particular technology depends on several factors.

To start with, Gremlin has recently introduced a new release. The whole implementation was rewritten to be language-agnostic. It, in turn, led to breaking changes that required the migration of consumer software systems. However, the Gremlin development team occasionally uploads relevant migration guides with numerous use-case examples [Tin21b]. Therefore, a comprehensive migration guide is expected to accompany every major update of Gremlin. In contrast, Shacl is a relatively new technology standard that did not experience any major updates yet. However, Shacl is actively gaining popularity and might experience more changes in the future to meet the requirements of its clients.

Comparing the completeness and comprehensiveness of these tools, both Gremlin and Shacl can feature extensive documentation. Gremlin has a complete manual and vast community support. Shacl also has a detailed specification and a growing community that provides support through forums, mailing lists, and GitHub issues.

Being a mature tool with a long history, Gremlin is a widespread query language with a well-established community. Hence, it is assumed that more resources and a large pool of experienced users are available. Nevertheless, the use of Shacl among developers is steadily increasing, and it is expected to gain more popularity in the coming years. As more developers become familiar with the power and flexibility of Shacl, a large and active community of users will likely emerge around this technology.

In summary, it is tough to definitively claim what tool is more advantageous in terms of maintainability. However, taking into account the maturity of Gremlin, its large community, and extensive documentation, it may be a more stable and reliable choice for a long-term software solution.

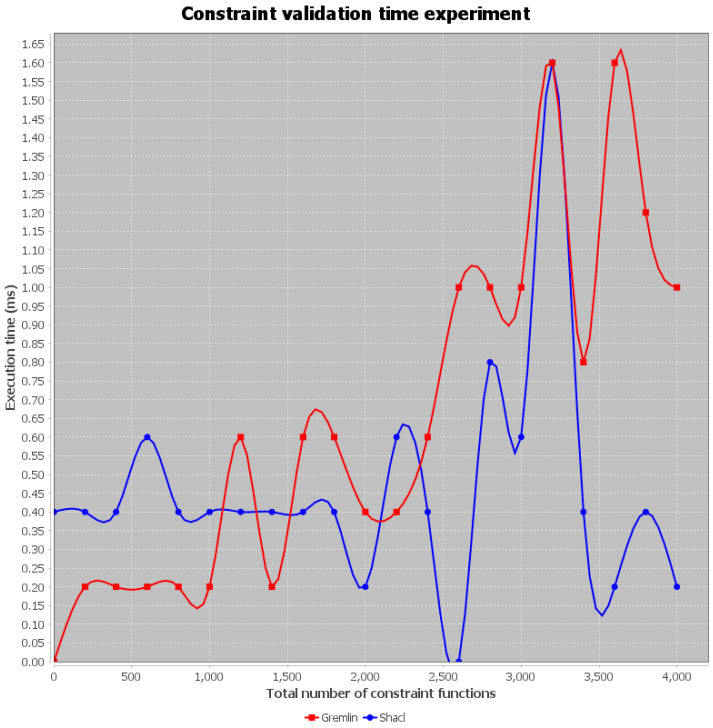


Figure 7.4: Constraint validation time

7.3.5 Interoperability

Interoperability is an important characteristic that denotes how easily a tool can incorporate with other systems and technologies.

Shacl is an accepted web technology data format to define the structure of resource graphs. It can be integrated with other RDF-based systems like RDF databases, SPARQL, and other graph data models [Knu+21].

Unlike Shacl, which is designed explicitly for validating RDF data and can be easily embedded into other RDF-like systems, Gremlin may require more effort to integrate with software that is not natively built on top of it. This is because Gremlin is not bound to any particular technology and is not accepted as a uniform standard for traversing graphs. For instance, Apache TinkerPop is an open-source graph computing framework that offers a range of tools for managing graph databases. Its primary query language is Gremlin [Fou23]. However, some other graph databases [; Pront] do not support Gremlin out-of-the-box, so integrating Gremlin into them may require additional configuration efforts.

To summarize, while Gremlin serves as a flexible and widely-adopted graph traversal language, its integration into systems that lack native support may entail additional complexity. On the other hand, Shacl is specifically suited for working with RDF data and can be more seamlessly integrated with other RDF-based systems.

7.4 Summary

Based on the implementation and evaluation chapter, we proved that Gremlin and Shacl could successfully be part of the user-driven constraint modeling tool. However, none of them is a technology to address all the problems and requirements since they both have strengths and weaknesses. In particular, this chapter helped to come up with the following conclusions:

- **Tool expressiveness** - Gremlin is more expressive than Shacl regarding what constraint

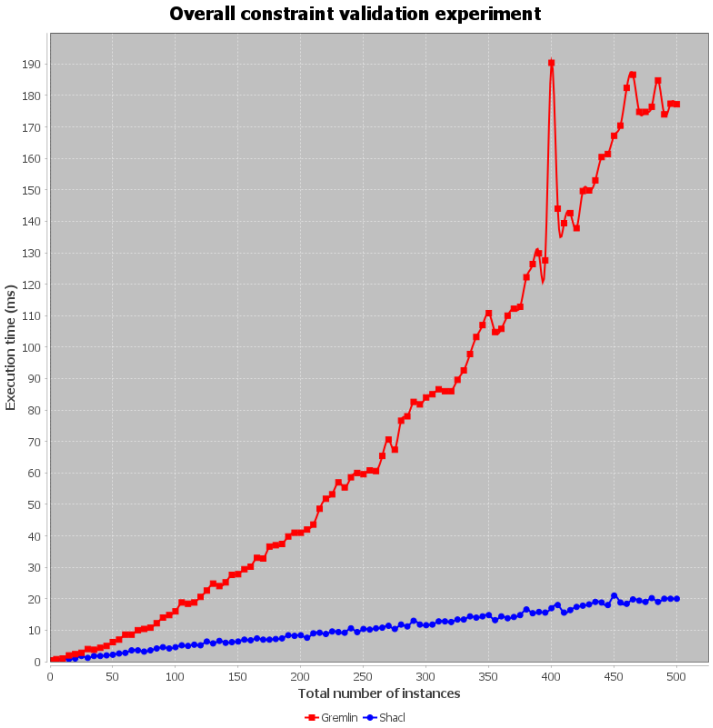


Figure 7.5: Overal constraint validation workflow

types could be constructed

- **Tool performance** - Shacl outperforms Gremlin on average by a factor of eight when mapping an abstract technical space to its platform-specific representation. The time to assess constraints is nearly the same
- **Tool extendability** - Both Shacl and Gremlin support mechanisms to dynamically create and execute new constraint types in addition to static ones
- **Maintainability** - Taking into account the well-established state of Gremlin, its comprehensive documentation, and other publicly available resources, a constraint framework might experience fewer challenges to maintain in comparison if it were implemented with Shacl
- **Interoperability** - Shacl is an accepted web standard to validate RDF graphs. In contrast, Gremlin is a query language among numerous other query languages. Therefore, embedding it into systems that don't have built-in support could involve extra complexities

Based on the points mentioned above, it is the responsibility of a developer to choose a tool that would suit better according to their specific prerequisites to implement a constraint modeling software system.

List of Figures

Bibliography

[] *Neo4j*. `https://neo4j.com/`. Accessed: March 16, 2023.

[AG16] Colin Atkinson and Ralph Gerbig. "Flexible deep modeling with melanee". In: *Modellierung 2016 - Workshopband*. Ed. by Stefanie Betz and Ulrich Reimer. Bonn: Gesellschaft für Informatik e.V., 2016, pp. 117–121.

[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.

[AGK15] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. "Opportunities and challenges for deep constraint languages". In: (2015).

[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+08] Gábor Bergmann et al. “Incremental pattern matching in the VIATRA model transformation system”. In: *Proceedings of the third international workshop on Graph and model transformations*. 2008, pp. 25–32.

[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.

[Bon+19] Iovka Boneva et al. “Shape designer for ShEx and SHACL constraints”. In: *ISWC 2019-18th International Semantic Web Conference*. 2019.

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

[CA10] Yoonsik Cheon and Carmen Avila. “Automating Java program testing using OCL and AspectJ”. In: *2010 Seventh International Conference on Information Technology: New Generations*. IEEE. 2010, pp. 1020–1025.

[CD03] Michelle L Crane and Juergen Dingel. “Runtime conformance checking of objects using Alloy”. In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003), pp. 2–21.

[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.

[Che+09] Yoonsik Cheon et al. “Checking design constraints at run-time using OCL and AspectJ”. In: (2009).

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

[Cor12] Oracle Corporation. *Java Native Interface Specification*. `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`. [Online; accessed 5-March-2023]. 2012.

[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.

[Dav+16] Ankur Dave et al. “Graphframes: an integrated api for mixing graph and relational queries”. In: *Proceedings of the fourth international workshop on graph data management experiences and systems*. 2016, pp. 1–8.

[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.

[DG10] Juan De Lara and Esther Guerra. "Deep meta-modelling with metadepth". In: *International conference on modelling techniques and tools for computer performance evaluation*. Springer. 2010, pp. 1–20.

[Di +19] Alessio Di Sandro et al. "Querying automotive system models and safety artifacts with MMINT and VIATRA". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2019, pp. 2–11.

[DSC16] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. "Mogwaı: a framework to handle complex queries on large models". In: *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. IEEE. 2016, pp. 1–12.

[EL16] Fajar J Ekaputra and Xiashuo Lin. "SHACL4P: SHACL constraints validation within Protégé ontology editor". In: *2016 International Conference on Data and Software Engineering (ICoDSE)*. IEEE. 2016, pp. 1–6.

[FCA09] Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. "Design for an adaptive object-model framework". In: *Proceedings of the 4th Workshop on Models@ run. time, held at the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*. Citeseer. 2009.

[FCM18] Nicolas Ferry, Franck Chauvel, and Brice Morin. "Multi-layered Adaptation for the Failure Prevention and Recovery in Cloud Service Brokerage Platforms". In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE. 2018, pp. 21–29.

[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.

[Fou23] The Apache Software Foundation. *Apache TinkerPop*. `https://tinkerpop.apache.org/index.html`. [Online; accessed 15-March-2023]. 2023.

[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.

[Fra+19] Enrico Franconi et al. "

\hbox {OCL} _\textsf {FO} *OCLFO* : *first–orderexpressiveOCLconstraintsforefficientintegritycheckin*
In: *Software & Systems Modeling* 18.4 (2019), pp. 2655–2678.

[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.

[Fum+17] Juan Fumero et al. "Just-in-time GPU compilation for interpreted languages with partial evaluation". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2017, pp. 60–73.

[Gei+06] Rubino Geiß et al. "GrGen: A fast SPO-based graph rewriting tool". In: *Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings 3*. Springer. 2006, pp. 383–397.

[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.

[Gro06] Object Management Group. "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (QVT)". In: *Object Management Group*. 2006. URL: `https://www.omg.org/spec/QVT`.

[Gro18] Object Management Group. "Meta Object Facility (MOF) Core Specification Version 2.5.1". In: Object Management Group, 2018. URL: `https://www.omg.org/spec/MOF/2.5.1`.

[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.

[Hen+10] Atzmon Hen-Tov et al. "Dynamic model evolution". In: *Proceedings of the 17th Conference on Pattern Languages of Programs*. 2010, pp. 1–13.

[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.

[HR07] Christian Hein and Tom Ritter. "Global constraint checking at run-time". In: *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*. IEEE. 2007, pp. 59–68.

[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.

[Jac04] Daniel Jackson. "Alloy 3.0 reference manual". In: *Software Design Group* (2004).

[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.

[JBG17] Edgar Jakumeit, Jakob Blomer, and Rubino Geiß. "The GrGen .NET User Manual". In: *Universität Karlsruhe, Fakultät für Informatik, Institute für Programmstrukturen und Datenorganisation, Lehrstuhl Prof. Goos* 1 (2017).

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. "Grgen. net: The expressive, convenient and fast graph rewrite system". In: *International Journal on Software Tools for Technology Transfer* 12 (2010), pp. 263–271.

[Jou+08] Frédéric Jouault et al. "ATL: A model transformation tool". In: *Science of computer programming* 72.1-2 (2008), pp. 31–39.

[Kah+16] Nafiseh Kahani et al. "The problems with eclipse modeling tools: a topic analysis of eclipse forums". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016, pp. 227–237.

[Kan14] Dominik Kantner. "Specification and implementation of a deep ocl dialect". PhD thesis. 2014.

[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.

[Keg27] Karl Kegel. *Modicio*. `https://github.com/modicio/modicio`. accessed 2023, February 27.

[Kim+16] Kosaku Kimura et al. "An Evaluation of Multi-Level Modeling Frameworks for Extensible Graphical Editing Tools." In: *MULTI@ MoDELS*. 2016, pp. 35–44.

[KK17] Holger Knublauch and Dimitris Kontokostas. "Shapes constraint language (SHACL)". In: *W3C Candidate Recommendation* 11.8 (2017).

[Knu+21] Holger Knublauch et al. *SHACL - Shape Constraint Language*. W3C Recommendation 20 July 2021. [Online; accessed 15-March-2023]. World Wide Web Consortium (W3C), 2021. URL: `https://www.w3.org/TR/shacl/`.

[KPP06] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "The epsilon object language (EOL)". In: *European conference on model driven architecture-foundations and applications*. Springer. 2006, pp. 128–142.

[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.

[LA18] Arne Lange and Colin Atkinson. "Multi-level modeling with MELANEE." In: *MoDELS (Workshops)*. 2018, pp. 653–662.

[Ler+20] Dorian Leroy et al. "Runtime Monitoring for Executable DSLs". In: *The Journal of Object Technology* 19.2 (2020), pp. 1–23.

[LF98] David C Luckham and Brian Frasca. "Complex event processing in distributed systems". In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998), p. 16.

[LGC10] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. *A framework for deep meta-modelling*. 2010. URL: `https://metadepth.org/`.

[LGC15] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. "Model-driven engineering with domain-specific meta-modelling languages". In: *Software & Systems Modeling* 14.1 (2015), pp. 429–459.

[Li+21] Yao Li et al. "A Performance Evaluation of Spark GraphFrames for Fast and Scalable Graph Analytics at Twitter". In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 5959–5959.

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

[MK15] Simon McGinnes and Evangelos Kapros. "Conceptual independence: A design principle for the construction of adaptive information systems". In: *Information Systems* 47 (2015), pp. 33–50.

[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.

[PK21] Paolo Pareti and George Konstantinidis. "A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs". In: *Reasoning Web International Summer School* (2021), pp. 115–144.

[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.

[Pront] The Apache TinkerPop Project. *Apache TinkerPop Documentation: Hadoop Plugin*. current. URL: `https://tinkerpop.apache.org/docs/current/reference/#hadoop-plugin` (visited on 03/16/2023).

[PSL08] Eric Prud'hommeaux, Andy Seaborne, and Hewlett-Packard Laboratories. Jan. 2008. URL: `https://www.w3.org/TR/rdf-sparql-query/`.

[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.

[RTJ00] Dirk Riehle, Michel Tilman, and Ralph Johnson. "Dynamic object model". In: *Pattern Languages of Program Design* 5 (2000), pp. 3–24.

[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).

[Sán+19] Beatriz Sánchez et al. "On-the-fly translation and execution of OCL-like queries on simulink models". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2019, pp. 205–215.

[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.

[Šen19] David Šenkýř. "SHACL shapes generation from textual documents". In: *Workshop on Enterprise and Organizational Modeling and Simulation*. Springer. 2019, pp. 121–130.

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

[ŠRN16] Martina Šestak, Kornelije Rabuzin, and Matija Novak. "Integrity constraints in graph databases-implementation challenges". In: *Proceedings of Central European Conference on Information and Intelligent Systems*. Vol. 2016. 2016, pp. 23–30.

[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.

[SVC22] Marco Stadler, Michael Vierhauser, and Jane Cleland-Huang. "Towards flexible runtime monitoring support for ROS-based applications". In: *RoSE'22: 4th International Workshop on Robotics Software Engineering Proceedings*. 2022.

[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.

[The23] The Apache Software Foundation. *SHACL Validation*. `https://jena.apache.org/documentation/shacl/index.html`. Accessed: March 9, 2023. 2023.

[Tik+18] Ulyana Tikhonova et al. "Constraint-based run-time state migration for live modeling". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 2018, pp. 108–120.

[Tin21a] Apache TinkerPop. *Gremlin Documentation*. Accessed: March 9, 2023. 2021. URL: `https://tinkerpop.apache.org/docs/current/reference/`.

[Tin21b] Apache TinkerPop. *Upgrade Guide*. `https://tinkerpop.apache.org/docs/current/upgrade/`. [Online; accessed 15-March-2023]. 2021.

[TPA12] Norbert Tausch, Michael Philippsen, and Josef Adersberger. "TracQL: a domain-specific language for traceability analysis". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 320–324.

[Vaj+11] Tamás Vajk et al. "Runtime model validation with parallel object constraint language". In: *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*. 2011, pp. 1–8.

[Var16] Dániel Varró. "Incremental queries and transformations: From concepts to industrial applications". In: *International Conference on Current Trends in Theory and Practice of Informatics*. Springer. 2016, pp. 51–59.

[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.

[YJ02] Joseph W Yoder and Ralph Johnson. "The adaptive object-model architectural style". In: *Software Architecture: System Design, Development and Maintenance* (2002), pp. 3–27.

[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.