e-Informatica Software Engineering Journal
Volume 00, Issue 0, pages: 1–57

BibTeX

Constraint Annotation Patterns for Prototype Generation within Domain-Driven Design

Van-Vinh Le, Duc-Hanh Dang*
*Corresponding author: hanhdd@vnu.edu.vn

Article info

Keywords:
Domain-Driven Design
Domain-Specific Language
UML/OCL
MPS
Annotation-Based Mechanism
Pattern.

Submitted: 1 Dec. 2025
Revised: 1 Jan 1970
Accepted: 1 Jan 1970
Available online: 1 Jan 1970

Abstract

Context: In domain-driven design (DDD), the domain model provides the foundation for business knowledge and architectural guidance, with OCL constraints ensuring the precise specification of business rules. Current DDD methods employ annotation-based domain-specific languages (aDSLs) to encode the domain model within the implementation, using annotations to specify its structure and logic. However, such aDSLs often fall short in capturing and integrating complex OCL constraints.
Objective: The focus of this paper is to introduces an extension of the annotation-based mechanism and specification patterns for aDSLs to express complex OCL constraints. Each OCL constraint in the domain model is associated with a corresponding specification pattern, referred to as a Constraint Annotation Pattern (CAP). These patterns employ annotation structures to represent the OCL constraints and capture their semantics, thereby enabling verification on the domain model aims to achieve the important features of DDD: expressiveness and effectiveness.
Method: Our approach involves specifying and managing CAPs. In this work, we define an initial catalog of CAPs, which enables the expression and integration of OCL constraints into domain models, thus producing an executable and unified domain model. To demonstrate our method, we develop a catalog of CAP patterns incorporated into DCSL, along with a supporting tool to evaluate CAP patterns through a case study to show that it is expressive and practical in real-world DDD projects.
Results: This paper presents two contributions. Firstly, it extends of aDSL/DCSL with a mechanism to represent complex OCL constraints, where each constraint is linked to a corresponding CAP. Secondly, it defines of a catalog of CAPs, in which each CAP encapsulates: (i) a UML structural perspective, (ii) an OCL specification, and (iii) annotations embedding OCL semantics into models.
Conclusions: The approach bridges the gap between the problem domain and the technical space while preserving the expressiveness of DDD, enabling automatic enforcement, reuse, and seamless model-to-code transformation.

1. Introduction

Domain-Driven Design (DDD) [1] is an approach to developing complex software systems that places the primary focus on the domain and its underlying logic. The core of DDD lies in the construction of a domain model that serves as the foundation for business knowledge and provides architectural guidance, with OCL constraints [2] ensuring the precise specification of business rules. The domain model guarantees *feasibility*, enabling the seamless transition between the model and the code, so that the model can be executed, refined, and evolved alongside the implementation, thus supporting applicability across various business contexts. It also guarantees *satisfiability* by capturing and fulfilling all business requirements and critical constraints defined by stakeholders, expressed through a ubiquitous language consistently used by domain experts and developers throughout analysis, design, and implementation.

The domain models [3–7] are usually represented in one of the following ways. First, UML/OCL class diagrams are used to capture domain concepts and their relationships [8]. Although they can be translated into executable models or object-oriented programs, a semantic gap remains between model representations and programming languages. Second, external DSLs [9–12] introduce custom syntaxes that enable precise specification of business logic, thereby facilitating constraint definition, automation, and effective communication between domain experts and developers. They are typically defined through metamodelling, which provides an explicit abstract syntax and supports validation via OCL. However, external DSLs require substantial effort to develop and maintain executable toolchains, including parsers, editors, and code generators, which may limit their adoption in agile DDD contexts. Recent DSLs, such as B-OCL [13], offer capabilities for structural modeling and limited constraint support, yet challenges remain in translating UML/OCL to executable code, integrating runtime OCL validation, and preserving lifecycle correctness. Finally, internal DSLs [14–16], especially annotation-based DSLs, embed domain logic within host languages like Java [17], enabling automatic test generation and verifiable implementations. Existing annotation-based approaches [4, 5, 18, 19] address only subsets of OCL, leaving comprehensive representation of complex constraints and fully automated code generation unresolved. Their main limitation lies in satisfiability, as host-language syntax restricts the natural representation of complex domain concepts.

Our previous works, DCSL [20] and AGL [21], employ an annotation-based DDD approach (aDSL) to construct specific domain models—both structurally and behaviorally—with the aim of supporting executable domain models for particular problem domains. These models provide concise and understandable representations of domain concepts, constraints, and business rules that closely resemble programming language syntax. Annotations are integrated directly into the source code, establishing a strong connection between domain model design and implementation. This method bridges the gap between the problem domain and the technical space by enabling the automatic generation of software artifacts directly from the domain model. Aligning domain concepts with their technical implementations reduces errors from manual programming and accelerates development.

However, these approaches remain limited, as aDSLs often fall short in capturing and integrating complex OCL constraints, which are essential for the precise specification of business rules in DDD [1, 2], and in specifying domain concerns and composing them within the executable unified domain model for automatic prototype generation, as introduced in UDML [22].

In this paper, we propose an extension of annotation-based mechanisms and specification patterns for aDSLs to express complex OCL constraints in domain models within the DDD paradigm. Each OCL constraint is mapped to a corresponding specification pattern, referred to as a Constraint Annotation Pattern (CAP), which employs annotation structures to represent constraint semantics and enables verification directly on the domain model. Furthermore, our approach supports the composition of aDSLs within an executable unified domain model, which serves as the foundation for automatic prototype generation, ensuring that both constraints and concerns are seamlessly integrated into software artifacts. By leveraging reusable CAP and concern composition mechanisms, the approach bridges the gap between the problem domain and the technical space, facilitating the automatic generation of software artifacts while preserving the expressiveness of DDD. The methodology is evaluated through detailed case studies, focusing on expressiveness, required coding effort, and applicability in real-world DDD projects.

Our methodology makes three key contributions:

1. Extension of aDSL/DCSL with a mechanism to represent complex OCL constraints, where each constraint is linked to a corresponding CAP.
2. Definition of a catalog of CAPs, in which each CAP encapsulates: (i) a UML structural perspective, (ii) an OCL specification, and (iii) annotations embedding OCL semantics into models, enabling automatic enforcement, reuse, and seamless model-to-code transformation.
3. Development of a supporting tool for DCSL with CAP, leveraging JetBrains MPS for meta-programming and systematic concern composition, and integrating with the jDomainApp framework for automated artifact generation. Evaluation on the CourseMan, OrderMan and ProcessMan case studies demonstrated both expressiveness and effectiveness in capturing complex OCL constraints, thereby validating the proposed methodology.

The rest of the paper is organized as follows: Section 2 presents our motivating example and the technical background. Section 3 describes the approach in detail. Section 4 presents the method of constraint annotation patterns. Section 5 presents support tools and experiments. Section 6 evaluates and discusses the issues surrounding the method. Section 7 discusses threats to the validity of our work. Section 8 reviews the related work, while Section 9 concludes the paper with a summary of the findings and future directions.

2. Motivating Example and Background

This section motivates our work through an example and provides the necessary background.

2.1. Motivating Example

Our study focuses on a course management system (CourseMan), in which the problem domain is captured by the domain model shown in Fig. 1. The domain concepts of this domain correspond to the classes: The class `Student` represents individuals enrolled at the university. Each student may register for multiple `CourseOfferings`, which represent specific instances of a `CourseModule` delivered in a given `AcademicTerm`. A `CourseModule` defines the syllabus and learning objectives of a course, whereas a `CourseOffering` associates the module with an academic term and an `Instructor` responsible for teaching it. The

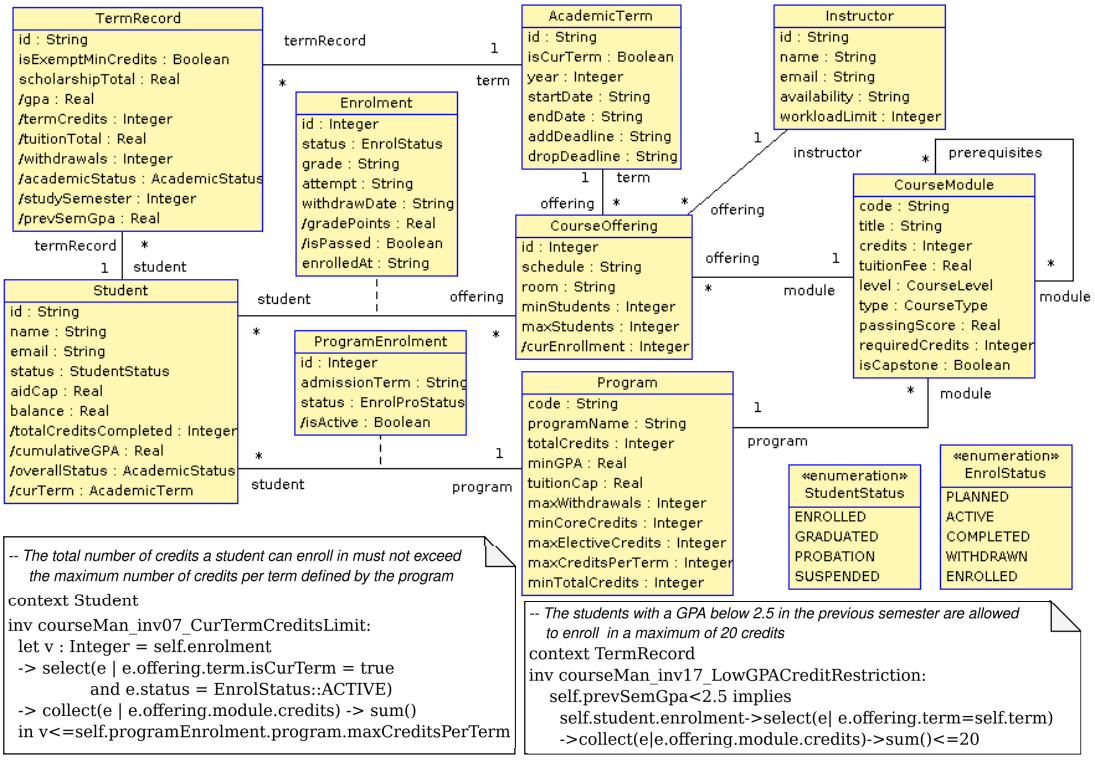


Figure 1: The **CourseMan** domain model represented as a UML/OCL class diagram.

`Program` class represents a degree program corresponding to an official academic structure that students enroll in and follow throughout their studies. A student's participation in a program is captured by `ProgramEnrolment`, while their enrolment in specific course offerings is managed through `Enrolment`, forming a many-to-many association between `Student` and `CourseOffering`. The `TermRecord` class captures each student's academic performance for a given term, summarizing enrolments, grades, and GPA calculations.

The UML class diagram in Fig. 1 captures the structural aspects of the CourseMan domain model. To specify behavioral and logical aspects that cannot be adequately captured using UML alone, we employ OCL constraints. One such constraint, illustrated in Fig. 1, ensures that the total number of credits a student enrolls in does not exceed the maximum per term specified by the student's program.

We aim to incorporate such a UML/OCL domain model into the context of DDD. This presents a challenge, as DDD encompasses not only model-centric aspects such as UML/OCL, but also code-centric and behavior-driven approaches. A domain model in DDD, as further explained in Subsection 2.2, must be technically feasible—that is, the model should be executable or interpretable as a program with well-defined operational semantics. Consequently, stakeholders, including both customers and developers, can be provided with appropriate views that are projected from the domain model.

A promising approach to this challenge, introduced in our previous work [20], is to define an annotation-based domain-specific language, as further explained in Subsection 2.3, for specifying domain models. The core idea is to use annotations to re-express OCL constraints. However, the current approach is limited to handling only essential constraints within domain models. To be more broadly applicable, it must also support more complex OCL constraints. Addressing this limitation serves as a key motivation for the present work.

2.2. Brief Overview of Domain-Driven Design

Evans' domain-driven design (DDD) methodology addresses the challenges of developing complex software systems by placing the domain at the center and providing both strategic and tactical means for managing complexity. DDD emphasizes building software models that closely reflect actual business logic, thereby ensuring maintainability and extensibility. Approaches that automatically generate domain models from high-level specifications—such as UML/OCL diagrams—enhance domain model representation and support the automation of the software development process, including source code and artifact generation [23, 24].

Building on this foundation, several studies [3–5] apply DDD to real-world business domains, aiming to accurately reflect business requirements while maintaining technical feasibility. Additionally, research [6, 7] focuses on constructing rich domain models that encapsulate business logic and address complex requirements, such as implementing business rules, workflows, and constraints in a DDD-compliant manner, while supporting business logic validation.

Three key features of DDD include: (1) feasibility—the domain model should be executable as code and vice versa, ensuring practical applicability across diverse business contexts; (2) satisfiability—the domain model must fulfill all business requirements and critical constraints as defined by stakeholders and expressed through a ubiquitous language [1]; and (3) effectiveness—the domain model should accurately describe current business requirements to ensure effective operation while supporting future expansion and maintainability. This ubiquitous language, shared among stakeholders including domain experts, designers, and developers, is developed iteratively through agile processes of domain requirement elicitation. Enhancing domain model representation and automating the software development process to satisfy these key DDD principles remains a central focus of current research.

2.3. DCSL: An Annotation-Based DSL to Specify Domain Models

Several contributions [12, 14–16] have explored the use of OCL in combination with annotations to represent domain models within a DDD framework, particularly for automatic test case generation. Meanwhile, domain-specific languages (DSLs) have been effectively applied to domain modeling and problem-solving in specific contexts, as demonstrated in prior research [9–12, 25]. Collectively, these studies illustrate how the flexibility and adaptability of DSLs can be leveraged to model diverse business domains.

Table 1: The essential structural constraints (adapted from [20])

Constraints	**Type**	**Descriptions**
Object mutability	Boolean	Whether or not the objects of a class are mutable [26]
Field mutability	Boolean	Whether or not a field is mutable (*i.e.* its value can be changed) [27]
Field optionality	Boolean	Whether or not a field is optional (*i.e.* its value needs not be initialised when an object is created) [27]
Field uniqueness	Boolean	Whether or not the values of a field are unique [27]
Id field	Boolean	Whether or not a field is an object id field [27]
Auto field	Boolean	Whether or not the values of a field are automatically generated (by the system) [27]

Constraints	Type	Description
Field length	Non-Boolean	The maximum length (if applicable) of a field (*i.e.* the field's values must not exceed this length) [27]
Min value of a field	Non-Boolean	The min value (if applicable) of a field (*i.e.* the field's values must not be lower than it) [27]
Max value of a field	Non-Boolean	The maximum value (if applicable) of a field (*i.e.* the field's values must not be higher than this) [27]
Min number of linked objects	Non-Boolean	The minimum number of objects to which every object of a class can be linked [8]
Max number of linked objects	Non-Boolean	The maximum number of objects to which every object of a class can be linked [8]

DCSL is an annotation-based domain-specific language (aDSL), introduced in [20], for specifying domain models. This aDSL enables concise and readable descriptions of domain concepts, constraints, and business rules, using a syntax that closely resembles that of the host programming language. DCSL supports the expression of OCL constraints [2]. In particular, annotations are used to capture and describe essential constraints in the domain model, as illustrated in Table 1.

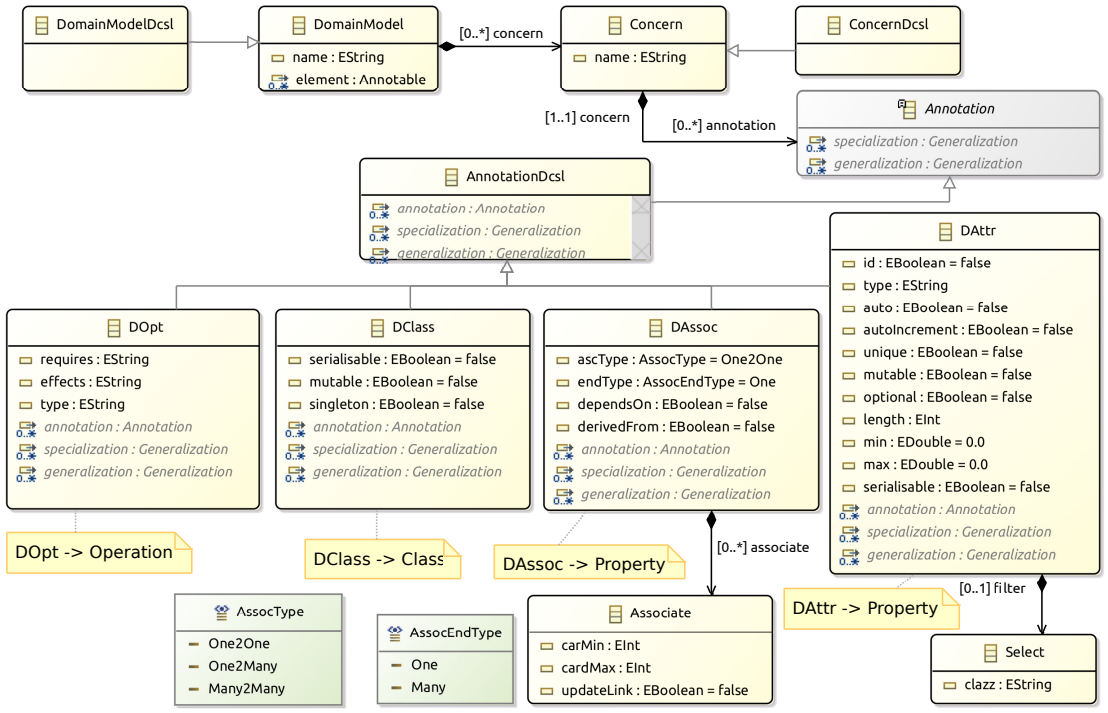


Figure 2: The DCSL metamodel.

Specifically, DCSL is defined by a metamodel as shown in Fig. 2, whose meta-concepts consist of core object-oriented programming language (OOPL) constructs and constraint-related elements. Specifically, the core meta-concepts include: the domain class `DClass`, the domain field `DAttr`, the associative field `DAssoc`, and domain method `DOpt`. The remaining meta-concepts, along with the properties of all concepts, are defined to express essential OCL constraints, which are summarized in Table 1.

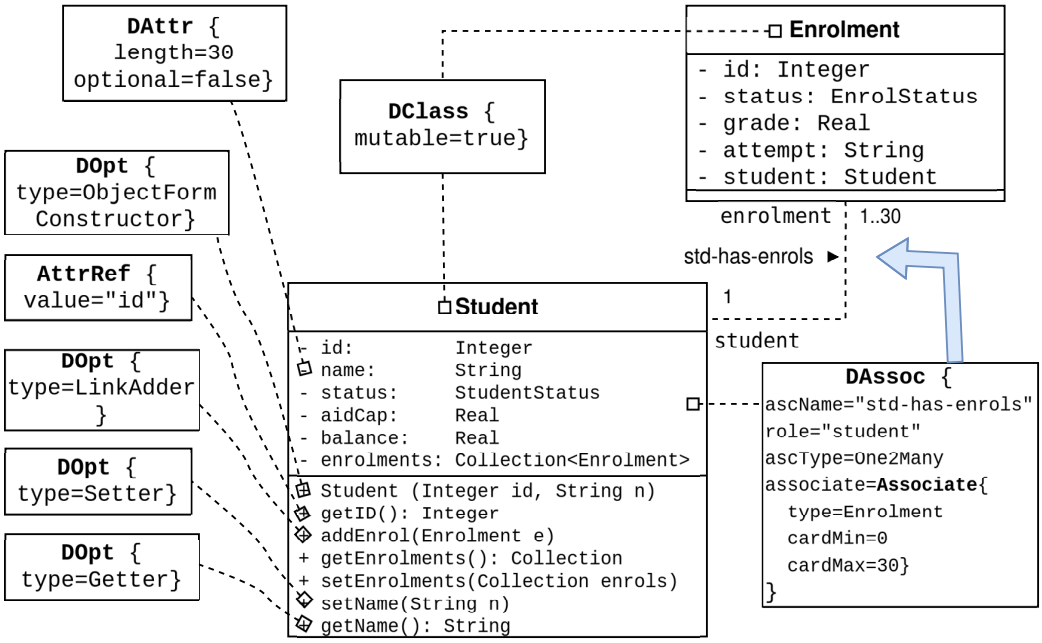


Figure 3: A simplified representation of the **CourseMan** domain model using DCSL.

Figure 3 illustrates a portion of the CourseMan domain model specified using the DCSL. This model includes two domain classes: `Student` and `Enrolment`, both annotated with the `DClass` element, indicating that they are mutable domain classes (`DClass.mutable`=true). Specifically, the `Student` class contains three domain fields: `id`, `name`, and `enrolments`. The `name` field is annotated with a `DAttr` element, specifying that it is mandatory (`DAttr.optional`= false) and has a maximum length of 30 characters (`DAttr.length`=30), among other attributes.

3. Overview of Our Approach

Figure 4 provides an overview of our approach, which consists of two main phases: *Phase 1* focuses on defining a catalog of constraint annotation-based patterns (CAPs) by eliciting and generalizing them from available domain requirements (Step 1 in Fig. 4). *Phase 2*, comprising the remaining three steps in Fig. 4, aims to derive a final software prototype that satisfies the input requirements of the target system. The main steps of our approach are summarized as follows.

Step 1 – Define CAPs: Based on a collection of domain requirements from different domains, this step aims to identify groups of OCL constraints, generalize each group, and derive a common pattern, called a CAP, to represent the OCL constraints belonging to that group. Each CAP encapsulates: (i) a UML class diagram to capture domain concepts and their relationships, (ii) an OCL specification of business rules and invariants, and (iii) annotations representing the OCL constraints.

Step 2 – Incorporate CAPs into the domain model: The input of this step consists of a set of OCL domain constraints and the existing catalog of CAPs. This step aims to

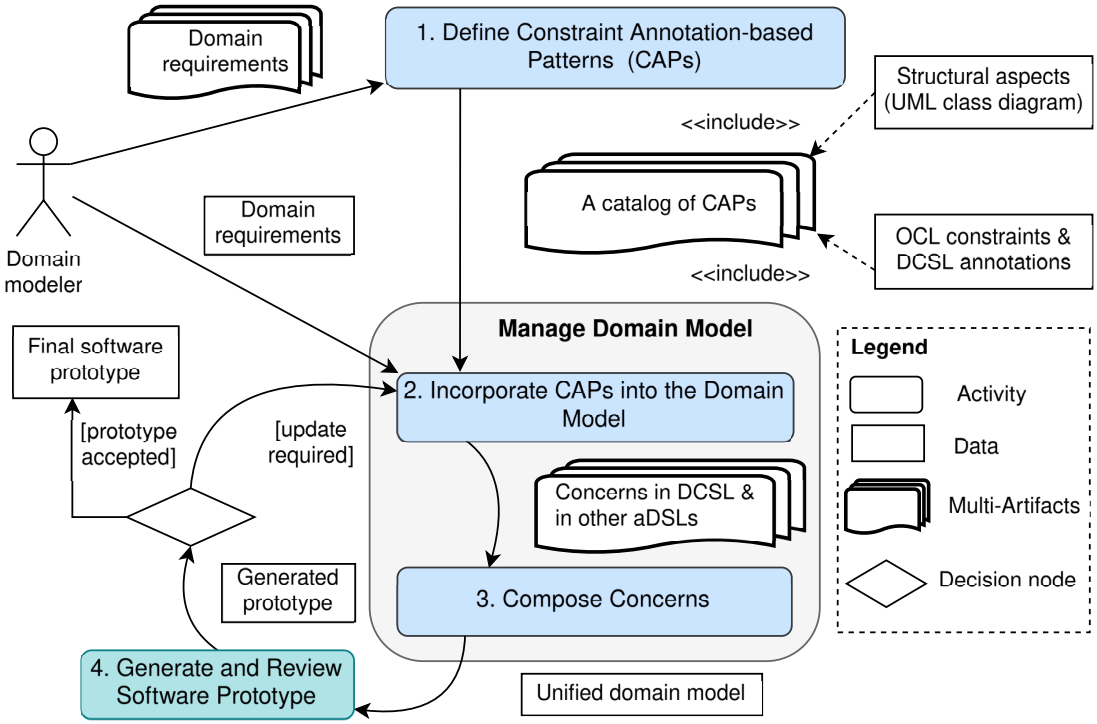


Figure 4: Overview of our approach.

represent the OCL constraints in terms of CAPs in order to constrain the domain model. Each CAP-based constraint is expressed through an extension of DCSL.

Step 3 – Compose concerns for a unified domain model: All relevant concerns covering both structural and behavioral aspects of the domain model, represented by aDSLs such as DCSL, are subsequently composed into a unified domain model.

Step 4 – Generate and review the software prototype: The unified domain model serves as the blueprint for the automatic generation of prototypes. The jDomainApp framework can be employed to perform this generation. The resulting prototype is then delivered to stakeholders for evaluation. If feedback is provided, the managed domain model is updated accordingly. The process then repeats, supporting a continuous improvement cycle until the software prototype satisfies the specified requirements.

4. Incorporation of CAPs into Domain Models

This section presents our method for specifying and managing CAPs. We extend DCSL to support the representation of CAP-based constraints. Furthermore, we define an initial catalog of CAPs, which enables the expression and integration of OCL constraints into domain models, thus producing an executable and unified domain model.

4.1. Syntax Aspect

The basic idea of a CAP is to use a template to represent OCL constraints within a domain model. Essentially, a CAP is a parameterized template. Each application of a CAP to represent a specific OCL constraint corresponds to assigning concrete values to the pattern's parameters. The input parameters of each template are defined through an annotation mechanism. We extend DCSL to represent CAPs, allowing each CAP to be formally specified and managed within the same modeling framework. From a syntactic perspective, each CAP pattern is specified with the following main components:

Pattern Name: Each CAP is assigned a unique identifier to facilitate its management and retrieval from the existing CAP catalog.

Description: Provides a brief explanation of the purpose and semantics of the OCL constraints represented by the CAP.

Template: Describes the parameterized OCL expression template, which conforms to the OCL syntax and corresponds to a specific group of OCL constraints sharing the same structural form. In particular, it includes the following specification elements:

- *Class diagram structure* — describes the relevant portion of the class diagram that defines the context of the OCL template. This structure includes parameters that specify class names, attributes, and relationships.
- *OCL template* — represents the parameterized OCL expression of the pattern.
- *Annotation specification* — define the parameters through a structured annotation mechanism used in the OCL expression of the pattern.

Example: Provides a concrete illustration of how the CAP is applied.

4.2. An Example and the First CAP Catalog

In this section, we introduce a CAP pattern named SumConstraint to illustrate our proposed approach. We then describe the first catalog of CAPs that we have collected and specified from various domain sources.

Pattern Name: SumConstraint

Description: This pattern defines OCL constraints for controlling and managing quantities, values, or resources through aggregation operations. Specifically, this pattern ensures that the aggregate value stored in an entity always equals the total of its related components. It is used to prevent discrepancies between derived data and source data, thereby supporting reconciliation and auditing.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 5 (label I) shows a class diagram depicting three classes, their attributes, and the associations between them. This diagram defines the context of the underlying OCL constraint template.
- *OCL template:* Figure 5 (label II) illustrates the OCL template used to generate constraints that express the following restriction: The sum of the values of the attribute `__sumAttr__` over all objects `o` in a collection filtered from the set of `__ClassD__` objects associated with the current `__ClassA__` instance (`self`) is computed according to the filtering criterion defined by the variable `__filterAttr_-`

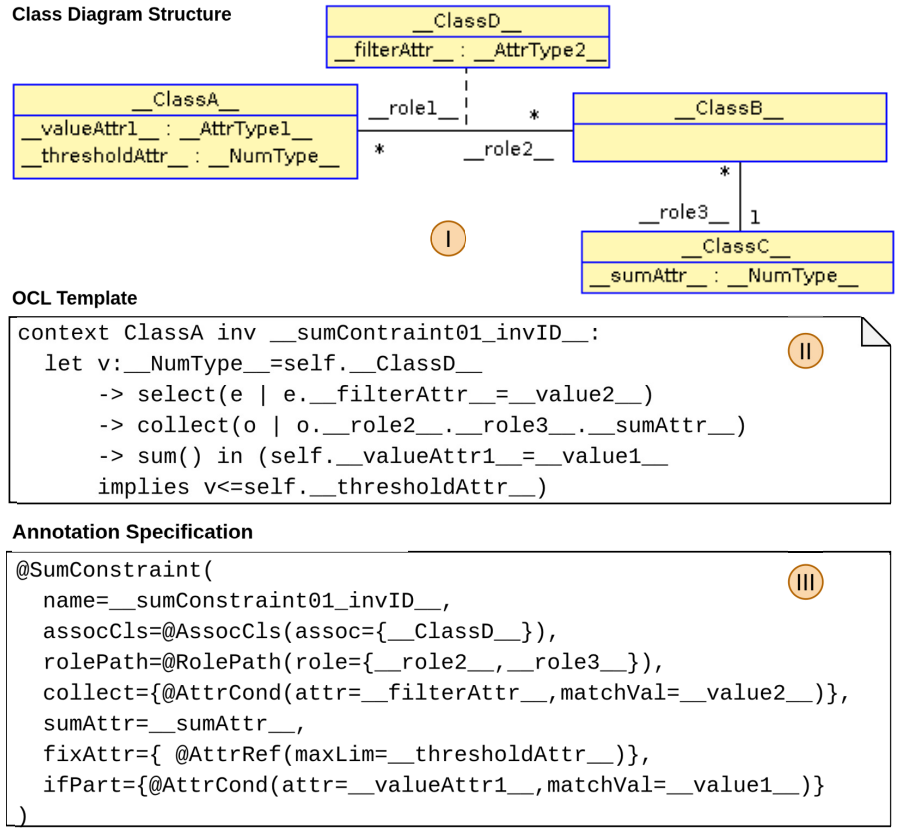


Figure 5: Template specification of the CAP pattern `SumConstraint`.

`_`. The computed sum is then validated against `__thresholdAttr__`, to ensure that the threshold condition is satisfied and that objects do not exceed the permitted limit. Note that the names of the variables in the template begin and end with "___". The variable name `__sumConstraint01_invID__` implies that this is OCL template type 01 of the SumConstraint pattern.

– *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 5 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model, as shown in Fig. 1, to illustrate this pattern. The underlying constraint ensures that the maximum number of credits a student can register for is limited to 12 credits per semester when the student is under academic warning. This helps the student focus on improving their GPA. The constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv06_ProbationRules:
  let v:Integer=self.enrolment
    ->select(e|e.status=EnrolStatus::ACTIVE)
    ->collect(e|e.offering.module.credits)->sum()
  in self.overallStatus=AcademicStatus::PROBATION
    implies v<=12
```

We define the specification in DSCL to generate the constraint as follows:

```
@SumConstraint(
  name='courseMan_inv06_ProbationRules',
  assocCls=@AssocCls(assoc={'Enrolment'}),
  rolePath=@RolePath(role={'offering','module'}),
  collect={@AttrCond(attr='status',
    matchVal='EnrolStatus::ACTIVE')},
  sumAttr='credits',
  fixAttr={@AttrRef(maxLim = 12)},
  ifPart={@AttrCond(attr='overallStatus',
    matchVal='AcademicStatus::PROBATION')}
)
class Student {...}
```

The SumConstraint pattern contains several OCL template variants corresponding to different groups of OCL constraints. The OCL template illustrated in Fig. 5 is one such variant. For example, consider another OCL constraint on the domain model shown in Fig. 1, as described below. This constraint ensures that students with a GPA below 2.5 in the previous semester are allowed to enroll in a maximum of 20 credits.

```
context TermRecord
inv courseMan_inv17_LowGPACreditRestriction:
    self.prevSemGpa<2.5 implies
      self.student.enrolment->select(e| e.offering.term=self.term)
      ->collect(e|e.offering.module.credits)->sum()<=20
```

The OCL template of the SumConstraint pattern to generate this constraint is defined as follows:

```
@SumConstraint(
  name='courseMan_inv17_LowGPACreditRestriction',
  assocCls=@AssocCls(assoc={'Enrollments'}),
  rolePath=@RolePath(role={'student,'offering','module'}),
  rolePath2=@RolePath(role={'term','termRecord'}),
  collect={@AttrCond(attr='curTerm',matchVal='true')},
  sumAttr='credits',
  fixAttr={@AttrRef(maxLim=20)},
  ifPart={@AttrCond(attr='prevSemGpa',maxLim=2.5)}
)
class TermRecord { ... }
```

We identified a CAP catalog that includes ten patterns, each formally specified and managed within a unified modeling framework. Each CAP represents a family that captures the shared semantic intent of constraints of the same kind, generalizing multiple domain rules with similar structures. Each CAP includes a set of types that distinguish between different OCL template variants of the CAP pattern. The two example OCL invariants discussed in this section are generated by the two OCL templates of the SumConstraint pattern. A comprehensive description of all the CAP patterns in our catalog is available in the Appendix A.

4.3. Semantics Aspect

The semantics of CAP directly correspond to those of OCL, as each CAP instance corresponds one-to-one with an OCL invariant in the domain model. This direct correspondence guarantees that evaluating any CAP constraint produces the same logical outcome as evaluating its OCL counterpart, thereby preserving OCL semantics within the DCSL.

Definition 1 (**CAP Application**)**.** Let CD be a class diagram representing the domain model; Let OCL_{CD} denote the set of all valid OCL invariants expressible over CD; Let CAP_i be a CAP defined by a parameterized OCL template $\mathcal{T}_i$ with parameter space $\mathcal{D}_i$. A *CAP application* on CD is the process of binding a parameter assignment $d_i \in \mathcal{D}_i$ to $\mathcal{T}_i$ so that the OCL template of the CAP becomes a valid OCL invariant on CD. Formally, this process is represented by the *generation mapping*

$$\mathcal{G} : (CAP_i, d_i) \longmapsto oclInv \in OCL_{CD},$$

where $\mathcal{G}(CAP_i, d_i)$ denotes the specific OCL invariant generated by instantiating the CAP template $\mathcal{T}_i$ with parameters d_i.

Example. Consider the constraint courseMan_inv06_ProbationRules. This constraint is generated by applying the CAP SumConstraint, as explained in Subsection 4.2. The corresponding generation mapping is defined as follows.

- CD represents the domain model CourseMan, as depicted in Fig. 1.
- CAP_i is the pattern `SumConstraint` in which:
  - $\mathcal{T}_i$ is the template of the CAP as shown in Fig. 5.
  - $\mathcal{D}_i$: The parameter space defined on CD; $d_i \in D_i$ denotes the parameter assignment that corresponds to the annotation specification for the invariant `courseMan_inv06_ProbationRules`, as detailed in Subsection 4.2.

Definition 2 (**CAP Catalog Completeness**)**.** Let $\mathcal{C}_{CAP}$ be a set of CAPs. The $\mathcal{C}_{CAP}$ is said to be *complete* with respect to the CD domain model if, for every invariant $oclInv \in OCL_{CD}$, there exists a $CAP_i \in \mathcal{C}_{CAP}$ and a parameter assignment $d_i \in \mathcal{D}_i$ such that $\mathcal{G}(CAP_i, d_i) = oclInv$.

Figure 6 shows an extension of the DCSL metamodel, as shown in Fig. 2, to represents CAPs. Each CAP is captured by a corresponding metaclass, including `SumConstraint` and `PrerequisiteConstraint`. The metaclass inherits the metaclass `AnnotationDcsl` as an extension point of DCSL for CAPs. Each CAP metaclass aggregates a set of metaclasses that encode the corresponding CAP templates. Each pattern template comprises parameters that are specified by the metaclasses associated with it. For example, the CAP `SumConstraint` comprises two templates, `SumConstraintTemplate01` and `SumConstraintTemplate02`, both of which share a common core defined by `SumConstraintTemplate`. The parameters of each template correspond one-to-one to the attributes of the `SumConstraint` annotation specified in Fig. 5.

4.4. Applying CAPs and Generating Software Prototypes

Following the proposed approach, the application of the CAP catalog and the generation of software prototypes are carried out through a three-phase process. Figure 7 illustrates this

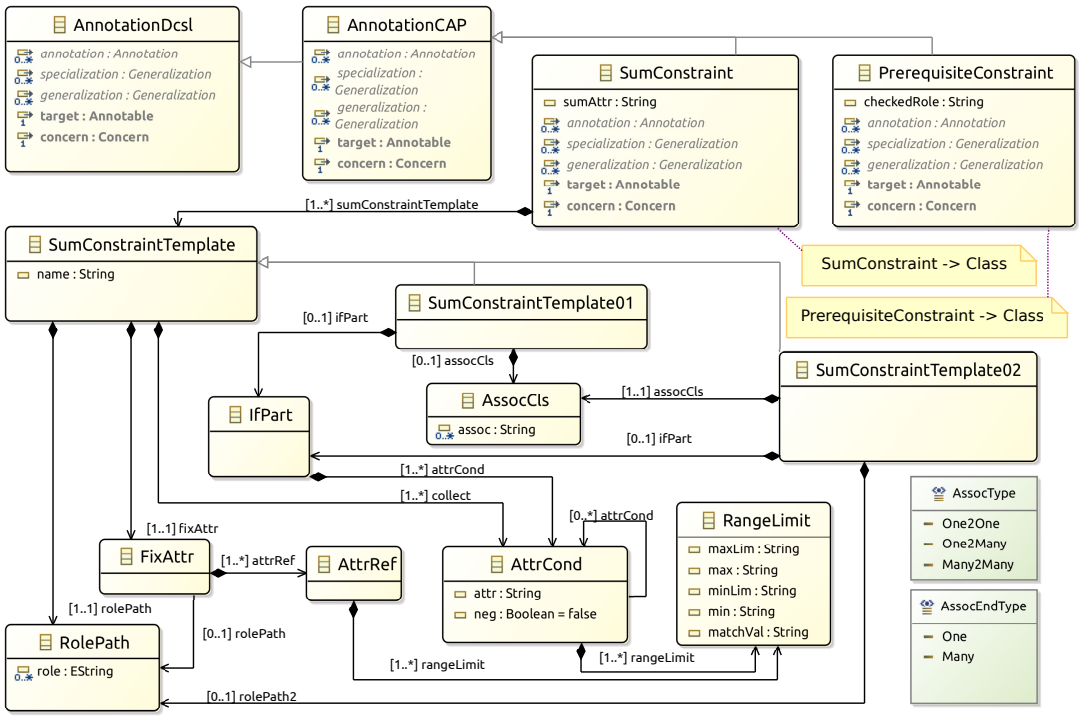


Figure 6: Extension of the DCSL metamodel to represent CAPs.

process, which involves employing and organizing the CAPs from the catalog to incorporate them into the unified domain model and to automate the generation of a software prototype.

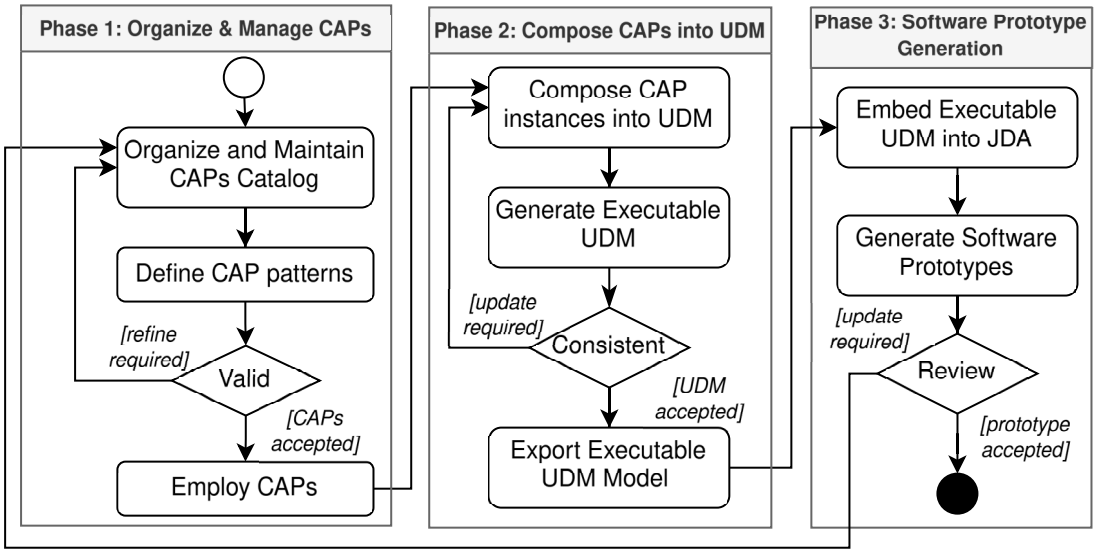


Figure 7: Incorporating CAPs into a unified domain model to obtain a software prototype.

Phase 1: Organization and Management CAPs

After the initial modeling phase, CAP instances are organized, classified, and managed to ensure consistency and reusability across the domain model. All defined CAPs are maintained within a structured *CAP Catalog*, which serves as a centralized repository of reusable constraint specifications and supports core management operations:

- *Insert* – registers newly defined CAPs with associated meta-data, including domain scope, applicability, and version information;
- *Select* – retrieves suitable CAPs for a given modeling task through domain- and category-based indexing;
- *Refine* – evolves or updates existing CAPs based on stakeholder feedback and prototype evaluation; and
- *Compose* – combines multiple CAPs into higher-level composite patterns to address cross-cutting concerns (e.g., security, logging).

Phase 2: Composing CAPs into the Unified Domain Model

CAPs are subsequently composed and integrated into the unified domain model, expressed in DCSL. This phase is supported by an interactive, graphical drag-and-drop interface developed using projectional editing in JetBrains MPS [28], which facilitates the visual configuration and linkage of CAP elements.

Phase 3: Software Prototype Generation

The executable UDM, derived from the DCSL specifications, serves as the basis for prototype generation. The overall process, illustrated in Fig. 8, automates the generation of software prototypes, thereby facilitating rapid validation and continuous refinement of the modeled domain logic.

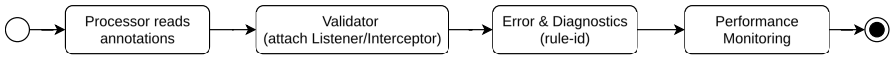


Figure 8: Overall process software prototype generation applied to the JDA framework.

5. Tool Support

In this section, we present the methodology for developing the supporting tool, called CAP/UDML. To compose CAPs into the UDM, we implemented CAP/UDML in JetBrains MPS, leveraging its projectional editing capabilities. The implementation is available as an open-source project on GitHub[1]

Specifically, Fig. 9 provides an overview of the CAP/UDML tool. On the left, label (1) shows the JetBrains MPS projectional editing environment for the CAP/UDML tool, which provides mechanisms for defining a unified domain model language to integrate with DCSL, as well as for composing them. This tool consists of two main components: First, the

[1]https://github.com/vinhskv/CAP_UDML/tree/main/designMPS

`Language` component defines the syntax of DSLs, including the UDML and its DCSL, via the `Structure`, `Editor`, `Constraints`, `Behavior`, and `Generator` aspects for generating Java code. Second, the `Solution` component defines UDML models—including DCSL and code generated by the MPS generator—while the `Sandbox` provides a testing environment for developing and validating UDML and CAP models.

On the right (label (2)), the graphical interface demonstrates how the CAP/UDML tool is applied to the CourseMan domain. As shown in label (3), the user first defines the core model in the `Sandbox`, and then creates CAP patterns that are composed into the Unified Domain Model (UDM) through an interactive drag-and-drop interface. This environment supports seamless model composition using classes, associations, and annotations according to the domain requirements, followed by automatic Java code generation for the JDA framework. A more detailed description is provided in the Appendix B.

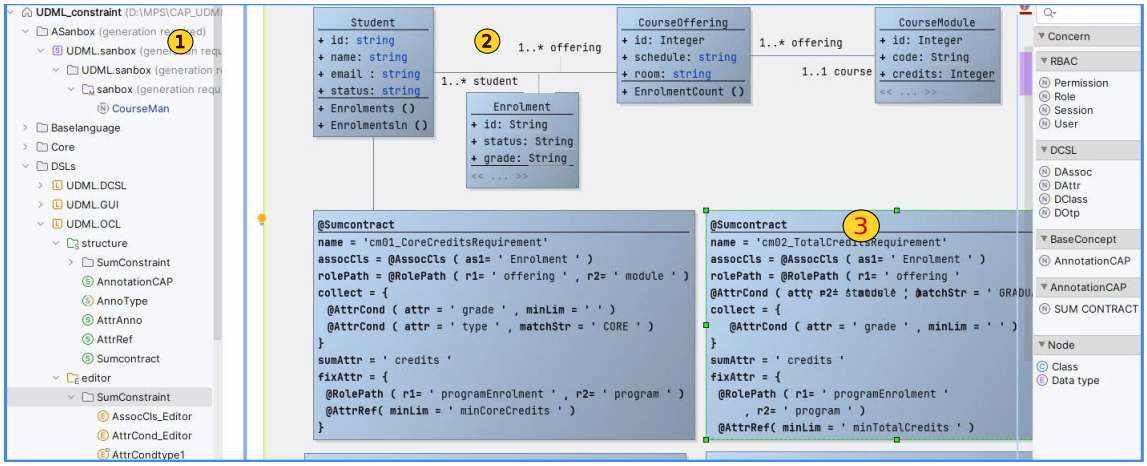


Figure 9: Supporting tool developed based on JetBrains MPS.

5.1. Software Artifact Generation

The implementation, illustrated in Fig. 10, is realized using Spring Boot and the JDA framework [29]. The framework can be accessedas the open-source project on GitHub[2]. The left panel of Fig. 10 shows the list of module classes and their corresponding domain classes in the structural model, each annotated with `@SumConstraint` and `@PrerequisiteConstraint` according to the CAP specification (as shown in the middle panel).

The generated CourseMan application with a graphical user interface (GUI) is presented in Fig. 11. This example demonstrates the runtime validation mechanism for enforcing the credit-limit constraint: the total number of registered credits for a student must not exceed 15. At the top of the figure, (1) shows a valid case where the student "Tran Van Anh" has registered for 14 credits, whereas (2) illustrates the validation error raised when attempting to exceed the credit limit. When performing the save operation to enroll in a new course, the system automatically activates the validation mechanism according to the method we proposed.

[2]https://github.com/vinhskv/CAP_UDML/tree/main/frameWorkGenCode

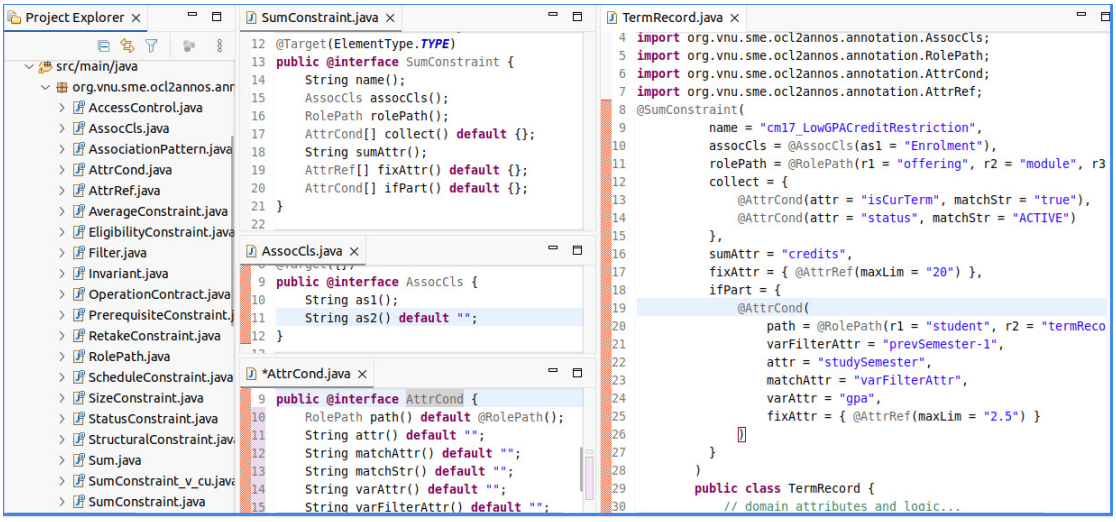


Figure 10: Tool support realization and usability of a CAP framework.

6. Evaluation and Discussion

This section presents the systematic evaluation results of CAP, accompanied by a discussion of its advantages and key findings. We use the criteria defined in [30] to evaluate CAP as a DSL.

6.1. Reseach Questions

Our evaluation focuses on assessing the expressiveness, required coding effort, and effectiveness of CAP, addressing the following research questions:

RQ1. How expressive are CAPs for modeling domain concepts compared to existing DDD methods?

RQ2: To what extent can CAP be applied in real-world software engineering practices?

6.2. Expressiveness of CAP in Domain Modeling

This section explains the results of our experiment, answering the raised research question:

RQ1. How expressive are CAPs for modeling domain concepts compared to existing DDD methods?

We consider the DCSL extension with integrated CAP support as a specification language, and evaluate it based on the following three criteria: expressiveness, required coding effort, and effectiveness. We compare CAP with our proposed DCSL[20] and commonly used third-party annotation sets, including OpenXAVA [19], Apache Causeway [18], and Actifsource [31].

Expressiveness. We analyzed their ability to use annotations for specifying and enforcing domain model aspects, thereby identifying language constructs and quantifying correspondences. Twelve evaluation criteria derived from annotation capabilities were applied to both our tool and the external tools. The evaluation employed three levels: fully supported, partially or indirectly supported, and not supported. Details of the pattern

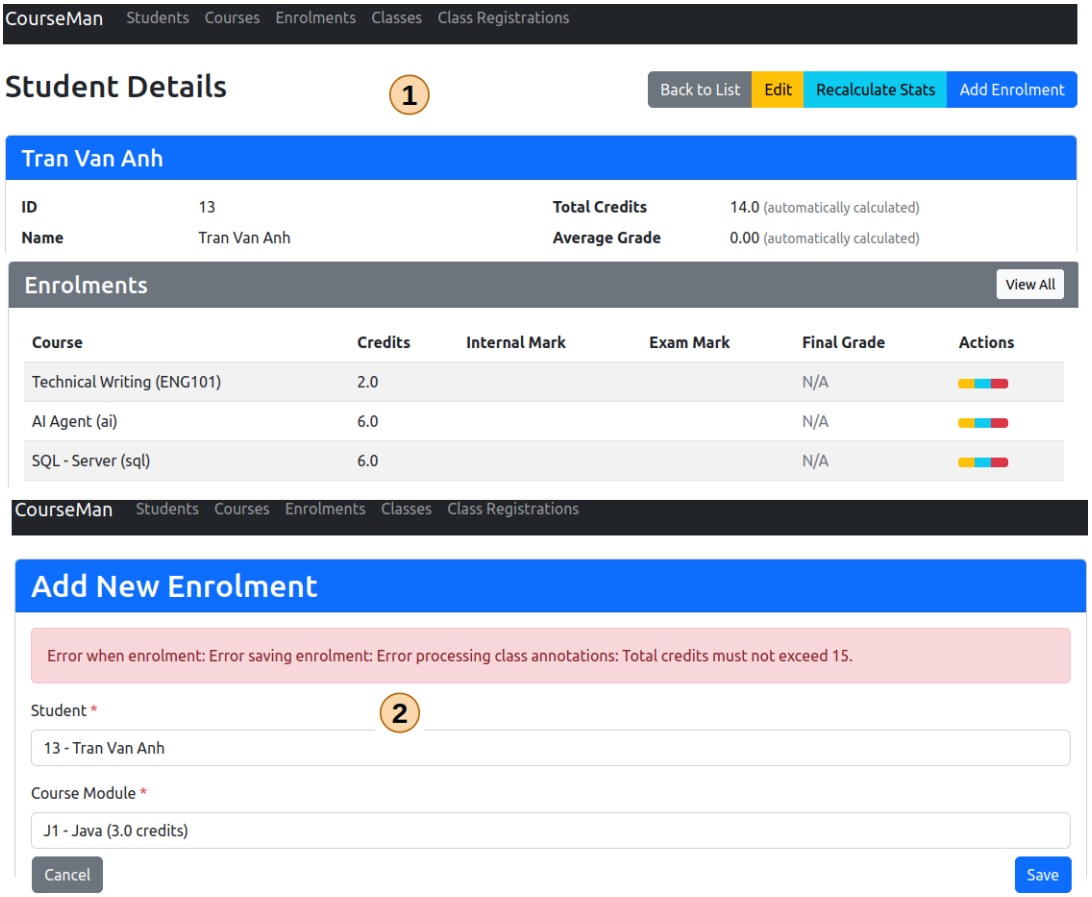


Figure 11: The GUI of the CourseMan software generated by the tool.

are provided in the Appendix C, with the summarized results presented in Table 2. The expressiveness of CAP ensures 100% satisfiability, meaning that all criteria are either fully supported or partially/indirectly supported (12/12 criteria), as domain models using CAP can capture all business constraints expressed in the ubiquitous language. In contrast, DCSL, OpenXAVA, and Apache Causeway still have 25% not supported (3/12 criteria). On the other hand, because Actifsource primarily supports structural modeling and code generation, its expressiveness for specifying and enforcing domain model aspects is also 100%, with all criteria being fully or partially/indirectly supported.

Table 2: The resulting comparison of tools based on expressiveness levels.

Expressiveness levels\ Compare the criteria	Fully supported	Partially indirectly supported	Not supported
CAP	10/12	2/12	0
DCSL	4/12	5/12	3/12
OPenXAVA	2/12	7/12	3/12
Apache Causeway	2/12	7/12	3/12
Actifsource	9/12	3/12	0

Required coding level. The criteria evaluate the extent of manual coding required to implement and enforce rules within the domain model, particularly for complex constraints:

1. *Automation:* Degree to which the tool automatically generates code, minimizing manual implementation effort.
2. *Complex constraint support:* Ability to handle advanced constraints (e.g., OCL expressions) without custom code.
3. *Model integration:* Ease of integrating constraints into the domain model with minimal manual adjustment.
4. *Learning curve:* Effort required for effective tool usage; a steep curve may increase manual coding.
5. *Flexibility:* Extent of customization or addition of domain logic without significant coding overhead.

We use three rating levels for the above five criteria: (++) reduces coding through automation and support for complex constraints via annotations, (+) reduces coding for basic cases but requires manual code for complex logic, and (−) manual implementation needed, tool provides structural scaffolding only. Details of the pattern are provided in the Appendix D, and the summarized results are presented in Table 3.

Table 3: Comparison of tools based on required coding level.

Tool\Criterion	**CAP**	**DCSL**	**Apache**	**OpenXava**	**Actifsource**
Automation of code generation	++	++	+	+	+
Support for complex constraints	++	-	-	+	+
Integration with domain model	++	+	+	+	+
Learning curve	++	++	+	+	-
Flexibility and customization	+	++	+	+	++

CAP and DCSL represent the most automated, constraint-driven approaches to domain modeling—they eliminate most manual validation code by combining annotation semantics and OCL/DSL generation. In contrast, Apache Causeway, OpenXava, and Actifsource remain partially manual frameworks, focused more on persistence and UI scaffolding than semantic constraint enforcement.

Effectiveness. The effectiveness of the tools is evaluated based on five key criteria that reflect the core aspects of DDD and model-based automation:

1. *Domain modeling capability:* The degree to which each tool supports the accurate representation of business-domain concepts and their interrelationships.
2. *Support for complex constraints:* The ability of the tool to express and enforce advanced domain constraints without requiring extensive manual coding.
3. *Consistency between model and code:* The extent to which the domain model is faithfully synchronized with the source code, aligning with the DDD principle that "the model is the code".
4. *Effectiveness in automation:* The level of automation achieved in generating source code, user interfaces, and validation logic, thereby reducing developer workload.
5. *Scalability and maintainability:* The capability of the tool to evolve, extend, and maintain the domain model consistently as system complexity increases.

Each criterion is evaluated using a three-level qualitative scale: (++) full support or superior performance; (+) sufficient or above-average support; (−) limited or basic support.

Show more detail in the Appendix E, the summarized results of this comparative evaluation are presented in Table 4.

Table 4: Comparative evaluation of tools based on domain modeling and automation criteria

Tool\Criterion	**CAP**	**DCSL**	**Apache**	**OpenXava**	**Actifsource**
Domain modeling capability	++	++	+	+	++
Support for complex constraints	++	-	+	-	+
Consistency between model and code	+	+	+	+	+
Effectiveness in automation	++	++	+	-	+
Scalability and maintainability	+	+	+	+	+

Both CAP and DCSL exhibit outstanding performance across all five criteria, achieving full automation in code generation and constraint enforcement through model annotations. CAP extends the DCSL approach by introducing reusable and parameterized constraint templates, enabling a direct one-to-one mapping between domain semantics and executable OCL invariants. This enhances both the expressiveness of the model and its consistency with the implementation. In contrast, OpenXAVA and Apache Causeway provide only partial automation—primarily focused on persistence and CRUD functionality—thus requiring developers to manually encode complex business rules. Meanwhile, Actifsource offers strong extensibility and meta-modeling flexibility, but its higher configuration complexity and reduced ease of maintenance limit its overall effectiveness in practical use.

6.3. CAP Applicability in Real-World Software Engineering

This section presents the results of our experiments on CourseMan, OrderMan and ProcessMan, addressing the research question raised:

RQ2: To what extent can CAP be applied in real-world software engineering practices?

To evaluate the applicability of the proposed CAP in real-world software engineering practice, we applied and assessed them across two domain-specific systems: CourseMan, OrderMan and ProcessMan. The results are summarized in Table 5.

Table 5: Constraint groups and CAP patterns

Constraint Group	**CAP catalog**	**Applicability of CAP**		
		CourseMan	OrderMan	Process-Man
Well-formedness constraints	DCSL in Table 1	11/11	11/11	11/11
Quantitative limits	SumConstraint	19/19	4/4	7/7
Dependency and prerequisite constraints	PrerequisiteConstraint	10/10	12/12	13/13
Scheduling and conflict constraints	ScheduleConstraint	17/17	6/6	10/10
Entity qualification constraints	EligibilityConstraint	20/20	5/5	12/12
Retry or reattempt rules	RetakeConstraint	8/8	2/2	8/8
Capacity constraints	SizeConstraint	17/17	3/3	15/15
Time and deadline constraints	TimeConstraint	10/10	6/6	13/13

Constraint Group	**CAP Catalog**	**Applicability of CAP**		
		CourseMan	OrderMan	Process-Man
Computation and derivation rules	SumProduct	15/15	8/8	8/8
Status-based constraints	StatusConstraint	12/12	6/6	15/15
Organizational structure constraints	StructuralConstraint	29/29	6/6	16/16
Pre-/post-condition constraints	✘	0/4	0/4	0/4
Access control constraints	✘	0/4	0/2	0/4
Automated reaction constraints	✘	0/4	0/2	0/4
	Total	**172/184**	**66/76**	**128/140**

Table 5 summarizes fourteen groups of OCL constraints. We propose a catalog of CAPs, consisting of multiple patterns that provide a syntactic and formally semantic framework for specifying business constraints in OCL, including structural validity, quantitative limits, dependencies, scheduling, eligibility, and organizational structure, etc,. However, three behavioral groups are excluded from the CAPs catalog: (1) *pre- and postconditions*, which require operational semantics and reasoning over pre- and poststates; (2) *access control*, which depends on platform-specific policies, roles, and authorization mechanisms; and (3) *automated reactions*, which involve event–task coupling, side effects, and execution ordering. These groups represent context-dependent behavioral constraints rather than structural invariants, are rarely reusable as invariant templates, and are highly architecture-dependent; therefore, they are omitted to maintain the conciseness and representativeness of the CAPs catalog. Experiments conducted on the CourseMan, OrderMan, and ProcessMan case studies demonstrate that the proposed CAPs can be expressed using annotation-based DSLs such as DCSL and integrated into a unified domain model to support automated generation and prototyping.

In the CourseMan case study, 184 constraints were examined and categorized into 14 groups. CAPs successfully represented 172 constraints across 11 groups, while 12 constraints fell outside its current scope due to dynamic conditions involving runtime states, events, or contextual triggers. These excluded cases mainly covered pre- and postcondition rules, access control policies, and automatic system responses, resulting in an overall coverage rate of 93.5%.

In OrderMan, 76 constraints were analyzed, spanning from structural integrity rules to complex business and behavioral logic. Among them, 66 were supported by CAPs, with the remainder exhibiting similar dynamic characteristics as those in CourseMan, yielding an applicability rate of 86.8%.

The ProcessMan evaluation followed a comparable trend: of 140 analyzed constraints, 128 were effectively captured by CAPs, while the rest involved dynamic or event-driven behaviors beyond the current framework's capability, achieving an overall applicability rate of 91.4%.

Beyond expressiveness and applicability, feasibility is validated through the generation and execution of executable prototypes directly from the unified domain model with CAPs, confirming that the proposed approach can be seamlessly realized as running software artifacts.

6.4. Discussion

We conclude our evaluation with several remarks on the design and implementation of CAP and its role in software design, and considerations related to expressiveness, required coding effort, effectiveness, and applicability.

In the design and implementation of CAP, the domain model provides a framework that bridges formal OCL constraints and executable Java code through annotation-driven syntax extensions. By transforming OCL invariants into reusable annotation-based patterns, CAP establishes a direct and traceable mapping from domain-level specifications to software implementation. Within software design, it extends DDD by embedding constraint definitions directly into the domain model, consolidating business rules in a single authoritative source, which reduces redundancy, mitigates inconsistencies, and lowers maintenance costs. Furthermore, the integration of the observable pattern enables continuous validation, ensuring immediate detection and resolution of constraint violations and thereby preserving semantic integrity throughout the system lifecycle. We discuss and point out the specific threats that affect our method as follows:

Expressiveness: CAP captures many OCL constraint patterns but has limitations with complex temporal logic, higher-order operations, and some quantifiers. Moreover, differences between OCL's precise semantics and Java's type system may lead to approximate enforcement.

Required coding level: CAP reduces manual validation code but requires developers to understand OCL concepts and the pattern mapping system. Initial setup of annotation processors and observer infrastructure entails overhead, though this is offset by reduced maintenance costs. Organizations must account for this learning curve.

Effectiveness: The effectiveness of CAP depends on performance and testing. Runtime reflection and constraint evaluation may affect scalability in large systems. While CourseMan showed good performance in educational scenarios, transaction-intensive systems may face bottlenecks. Effectiveness also relies on comprehensive test coverage.

Applicability: We defined and applied the CAPs catalog to CourseMan, OrderMan and ProcessMan, covering major business rules and constraints. Broader applicability requires extending the pattern library with domain-specific constraints and supporting runtime components (e.g., listeners, processors, validators). Current results demonstrate feasibility in representative DDD projects, while wider adoption will benefit from systematic pattern expansion and component development.

7. Threats to Validity

This section outlines potential threats to the validity of both our proposed method and its evaluation. Following the classification by Runeson et al. [32], we organize these threats into three groups: construct validity, internal validity, and external validity.

7.1. Construct Validity

In our case study, we assumed that the domain requirements were fully captured without misinterpretation that could result in an unsatisfactory model or inconsistent business rules. The modeler collected all domain requirements to ensure that the constructed models were accurate and satisfactory. Our method mitigated the threat of misinterpretation by enabling the explicit representation of domain class models, OCL constraints, and parameterized specification template annotations. These were then encapsulated into a CAP pattern, which facilitated their composition into an executable unified domain model within a domain-driven architecture.

7.2. Internal Validity

One threat arises from translating high-level domain specifications (UML class diagrams, OCL constraints, and annotations) into a CAP template. To address this, we provided guidelines in Section 4 for applying CAPs with the initial catalog. Another threat is that OCL constraints may not capture all input patterns. We mitigated this by evaluating three complex applications (CourseMan, OrderMan and ProcessMan) and by planning to extend the catalog with additional patterns. Finally, implementation errors may occur when composing CAP concerns into a unified domain model and applying the JDA framework. To reduce this risk, we thoroughly tested and reviewed the code and released it publicly for community validation.

7.3. External Validity

Threats to the external validity of our method concern its applicability to DDD-based software. First, since the method targets AST-oriented modeling within DDD, its use may be limited in other architectural paradigms. Second, the representativeness of the case study could be questioned. Nonetheless, our pattern-based approach mitigates this risk by adhering to the well-established principle of keeping patterns generic.

8. Related Work

Several studies have focused on transforming class diagrams and business rules specified using OCL into executable code or annotations in OOPLs. For example, the work by [33] proposed a method for generating Java code from OCL constraints by translating them into SQL queries; however, their approach primarily targeted database validation rather than in-memory domain model constraints. Similarly, the work by [34] explored Java code generation based on OCL rules, although it generates standalone validation methods rather than using annotations to embed constraints directly within the model.

The works by [35, 36] proposed a framework for translating OCL into Java annotations, but their solution required significant manual intervention when handling complex constraints. OCL2Java, a semi-automated tool, mainly supports simple OCL patterns such as and, or, not, but lacks support for complex patterns like exists, forAll, and set operations.

More recently, the studies by [12, 14–16] use OCL combined with annotations to represent domain models in a DDD approach. The study by [37] developed the OCL2MSFOL approach, which translates OCL to first-order logic for formal verification but does not address the implementation in domain models.

However, all the works have the following limitations: (1) Lack of full automation, i.e., most previous approaches require manual intervention or user refinements when dealing with complex constraints; (2) Limited pattern support, i.e., existing methods often only support simple model patterns and struggle to handle complex OCL, such as exists, forAll, or set operations on Class, Attribute, Association, and Operation by [36]; (3) Accuracy and efficiency issues, i.e., some approaches generate annotations that do not fully preserve the semantics of the original OCL constraints, particularly for quantifiers and collection operations. Others produce inefficient validation code that can lead to performance bottlenecks in large-scale systems; (4) Integration challenges, i.e., most existing solutions focus solely on constraint validation without addressing the derivation of calculated values or the propagation of changes through the domain model. This creates a disconnect between validation logic and the natural evolution of the model state; and (5) Tooling complexity, i.e., any frameworks require separate code generation steps and specialized tools, adding complexity to the development workflow and creating maintenance challenges when models evolve.

Annotation in annotation-based patterns enhances clarity and contextual understanding, improving collaboration, traceability, and domain alignment in complex adaptive systems.

9. Conclusion and Future Work

In this paper, we introduce a catalog of CAPs—an extension to DCSL that represents complex OCL constraints through pattern-based annotations covering syntax, semantics, and the organization and management of CAPs. By binding OCL specifications to reusable patterns, the approach bridges the gap between the problem domain and the technical space, enabling the automatic generation of software artifacts directly from the domain model. It (i) maintains continuous alignment between domain understanding and implementation, (ii) reduces manual coding effort and error risk through generated validators, and (iii) promotes reuse via a catalog of domain patterns. We implement CAP with a supporting toolchain that leverages JetBrains MPS to compose CAPs into a unified domain model and integrates with the JDA framework to automate artifact generation. Our approach advances the state of the art in DDD by bridging the gap between model and code while addressing the two central aims of DDD—feasibility and satisfiability.

Future work will continue to refine CAP to apply to more other domain problems and as a library of diverse patterns for complex OCL expressions, ensuring flexibility, ease of integration, high performance, compliance with common standards, and ease of maintenance. Furthermore, we aim to complete the development of a common architecture to map model patterns to multiple OOPLs, which is also part of our future work.

CRediT authorship contribution statement

Van-Vinh Le: Conceptualization, Methodology, Writing- Original draft preparation, Investigation, practical in real-world DDD projects, Validation, Writing- Reviewing and Editing. Duc-Hanh Dang: Conceptualization, Methodology, Formal analysis, Validation, Writing-Reviewing and Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[2] OMG, *Object Constraint Language 2.4*. OMG, 2014. [Online]. https://www.omg.org/spec/OCL/2.4/PDF

[3] V. Vernon, *Implementing domain-driven design*, 1st ed. Addison-Wesley Professional, 2013.

[4] O. Özkan, Babur, and M.v.d. Brand, "Domain-Driven Design in Software Development: A Systematic Literature Review on Implementation, Challenges, and Effectiveness," Nov. 2023. [Online]. http://arxiv.org/abs/2310.01905

[5] S.K. Jaiswal and R. Agrawal, "Domain-Driven Design (DDD)- Bridging the Gap between Business Requirements and Object-Oriented Modeling," *Int. Innovative Research in Engineering and Management*, Vol. 11, No. 2, 2024, pp. 79–83, number: 2.

[6] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed. Morgan&Claypool, 2017.

[7] J. Griffin, "Domain-Driven Laravel, Learn to Implement Domain-Driven Design Using Laravel," Jan. 2021.

[8] OMG, *Unified Modeling Language 2.5.1*. Object Management Group, 2017.

[9] M. Fowler, *Domain-Specific Languages*, 1st ed. Addison-Wesley Professional, Sep. 2010.

[10] Andrzej Wąsowski and Thorsten Berger, *Domain-Specific Languages Effective Modeling, Automation, and Reuse*. Springer Cham, 2023.

[11] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander et al., *DSL Engineering - Designing, implementing and using domain-specific languages*. Stuttgart, Germany: M Volter / DSLBook.org, 2013. [Online]. http://www.dslbook.org/

[12] I. Córdoba-Sánchez and J. De Lara, "Ann: A domain-specific language for the effective design and validation of Java annotations," *Computer Languages, Systems Structures*, Vol. 45, Apr. 2016, pp. 164–190.

[13] F.U. Haq and J. Cabot, "B-OCL: An Object Constraint Language Interpreter in Python," Mar. 2025, arXiv:2503.00944 [cs]. [Online]. http://arxiv.org/abs/2503.00944

[14] C. Noguera and L. Duchien, "Annotation Framework Validation Using Domain Models," in *Model Driven Architecture – Foundations and Applications*. Springer, Berlin, Heidelberg, 2008, pp. 48–62, iSSN: 1611-3349. [Online]. https://link.springer.com/chapter/10.1007/978-3-540-69100-6_4

[15] A.D. Brucker, M.P. Krieger, D. Longuet, and B. Wolff, "A Specification-Based Test Case Generation Method for UML/OCL," in *Models in Software Engineering*. Springer, Berlin, Heidelberg, 2011, pp. 334–348, iSSN: 1611-3349. [Online]. https://link.springer.com/chapter/10.1007/978-3-642-21210-9_33

[16] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L.C. Briand, "Generating Test Data from OCL Constraints with Search Techniques," *IEEE Transactions on Software Engineering*, Vol. 39, No. 10, Oct. 2013, pp. 1376–1402, conference Name: IEEE Transactions on Software Engineering. [Online]. https://ieeexplore.ieee.org/document/6491405

[17] J.G.B.J.G. Steele and G.B.A. Buckley, *The Java language Specificition*. Bhaskarjyoti Saikia, 2020.

[18] Dan Haywood, "Apache Isis - Developing Domain-Driven Java Apps," *Methods & Tools: Practical knowledge source for software development professionals*, Vol. 21, No. 2, 2013, pp. 40–59.

[19] J. Paniza, *Learn OpenXava by example*, 1st ed. CreateSpace, 2011.

[20] D.M. Le, D.H. Dang, and V.H. Nguyen, "On domain driven design using annotation-based domain specific language," *Computer Languages, Systems & Structures*, Vol. 54, Dec. 2018, pp. 199–235.

[21] D.H. Dang, D.M. Le, and V.V. Le, "AGL: Incorporating behavioral aspects into domain-driven design," *Information and Software Technology*, Vol. 163, Nov. 2023, p. 107284.

[22] V.V. Le and D.H. Dang, "An Approach to Composing Concerns for an Executable Unified Domain Model," in *2024 RIVF Int. Conf. On Computing and Communication Technologies (RIVF)*, Dec. 2024, pp. 424–428, iSSN: 2473-0130. [Online]. https://ieeexplore.ieee.org/abstract/document/11009109

[23] J. Sangabriel-Alarcón, J.O. Ocharán-Hernández, K. Cortés-Verdín, and X. Limón, "Domain-Driven Design for Microservices Architecture Systems Development: A Systematic Mapping Study." IEEE Computer Society, Nov. 2023, pp. 25–34. [Online]. https://www.computer.org/csdl/proceedings-article/conisoft/2023/288300a025/1Y7jTxtUNBm

[24] V.V. Le, N.T. Be, and D.H. Dang, "On Automatic Generation of Executable Domain Models for Domain-Driven Design," in *2023 15th International Conference on Knowledge and Systems Engineering (KSE)*, Oct. 2023, pp. 1–6, iSSN: 2694-4804. [Online]. https://ieeexplore.ieee.org/document/10299453

[25] A. Wąsowski and T. Berger, "Internal Domain-Specific Languages," in *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer, Cham, 2023, pp. 357–394.

[26] B. Liskov and J. Guttag, *Program development in Java: abstraction, specification, and object-oriented design*. Addison-Wesley Professional, 2000.

[27] J.A. Hoffer, J. George, and J.A. Valacich, *Modern Systems Analysis and Design*, 7th ed. Pearson, 2013.

[28] M. Voelter, "Language and IDE Modularization and Composition with MPS," Lecture Notes in Computer Science (LNPSE), R. Lämmel, J. Saraiva, and J. Visser, Eds. Springer, Berlin, Heidelberg, 2013, Vol. 7680, pp. 383–430.

[29] D.M. Le, D.H. Dang, and H.T. Vu, "jDomainApp: A Module-Based Domain-Driven Software Framework," in *Proc. 10th Int. Symposium on Information and Communication Technology (SoICT)*, Dec. 2019, pp. 399–406.

[30] Thakur and U. Pandey, "The Role of Model-View Controller in Object Oriented Software Development," *Nepal Journal of Multidisciplinary Research*, Vol. 2, Nov. 2019, pp. 1–6.

[31] *Actifsource*. Actifsource AG, Switzerland - all rights reserved., 2017. [Online]. https://www.actifsource.com/_downloads/ActifsourceManual_ActifsourceUserManual.pdf

[32] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, Vol. 14, No. 2, Apr. 2009, pp. 131–164.

[33] J. Cabot and E. Teniente, "Transformation techniques for OCL constraints," *Science of Computer Programming*, Vol. 68, No. 3, Oct. 2007, pp. 179–195. [Online]. https://www.sciencedirect.com/science/article/pii/S0167642307001256

[34] M. Rackov, S. Kaplar, M. Filipović, and G. Milosavljević, "Java code generation based on ocl rules," in *6th International Conference on Information Society and Technology*, 2016, pp. 191–196.

[35] L. Hamann, O. Hofrichter, and M. Gogolla, "OCL-Based Runtime Monitoring of Applications with Protocol State Machines," in *Modelling Foundations and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, Vol. 7349, pp. 384–399, series Title: Lecture Notes in Computer Science. [Online]. http://link.springer.com/10.1007/978-3-642-31491-9_29

[36] J. Chimiak-Opoka, B. Demuth, A. Awenius, D. Chiorean, S. Gabel et al., "OCL tools report based on the ide4OCL feature model," *Electronic Communications of the EASST*, Vol. 44, 2011. [Online]. https://eceasst.org/index.php/eceasst/article/view/1850

[37] C. Dania and M. Clavel, "OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. Saint-malo France: ACM, Oct. 2016, pp. 65–75. [Online]. https://dl.acm.org/doi/10.1145/2976767.2976774

[38] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, *Domain-specific languages in practice: with JetBrains MPS*. Springer Nature, 2021.

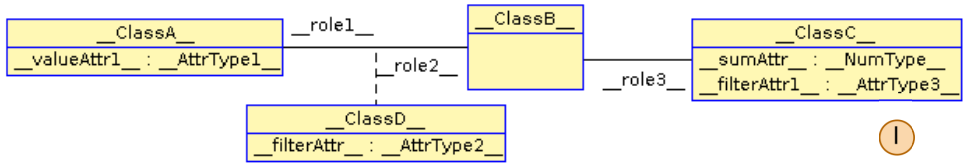
A. Catalog of CAP patterns

A.1. Types of the SumConstraint Pattern

Figure 5 depictes the type 01 of the first catalog CAP patterns (SumConstraint. In this section, we identify the remaining types—Equality/Reconciliation and Group-Wise/Relative—to distinguish the various OCL template variants associated with this pattern.

Type02 — Equality / Reconciliation (Summation match).

Class Diagram Structure



OCL Template

```
context ClassA inv __sumContraint02_invID__:
let total : __NumType__ =  self.__ClassD__
    ->select(e | e.__filterAttr__ = __valueTotal__)
    ->collect(o | o.__role2__.__role3__.__sumAttr__)->sum()
in let part1 : __NumType__ =  self.__ClassD__
    ->select(e | e.__filterAttr1__ = __valuePart1__)
    ->collect(o | o.__role2__.__role3__.__sumAttr__)->sum()
in let part2 : __NumType__ =  self.__ClassD__
    ->select(e | e.__filterAttr1__ = __valuePart2__)
    ->collect(o | o.__role2__.__role3__.__sumAttr__)->sum()
in (self.__valueAttr1__ = __value1__ implies total = part1 + part2)
```

II

Annotation Specification

```
@SumConstraint(
  name     = __sumConstraint02_invID__,
  assocCls = @AssocCls(assoc={ __ClassD__ }),
  rolePath = @RolePath(role={__role2__, __role3__}),
  collect={@AttrCond(attr = __filterAttr__, matchVal = __valueTotal__)},
  sumAttr= __sumAttr__,
  collect1={@AttrCond(attr =__filterAttr1__,matchVal =__valuePart1__)},
  sumAttr1 =__sumAttr__,
  collect2={@AttrCond(attr =__filterAttr1__,matchVal=__valuePart2__)},
  sumAttr2=__sumAttr__,
  fixAttr={@CompCond(val =sumAttr,matchVal=sumAttr1 + sumAttr2)},
  ifPart={@AttrCond(attr = __valueAttr1__, matchVal = "__value1__")}
)
```

III

Figure 12: Type 02 of the template specification of the CAP pattern `SumConstraint`.

– *Class diagram structure:* Figure 12 (label I) shows a class diagram depicting three classes, their attributes, and the associations between them. This diagram defines the context of the underlying OCL constraint template.

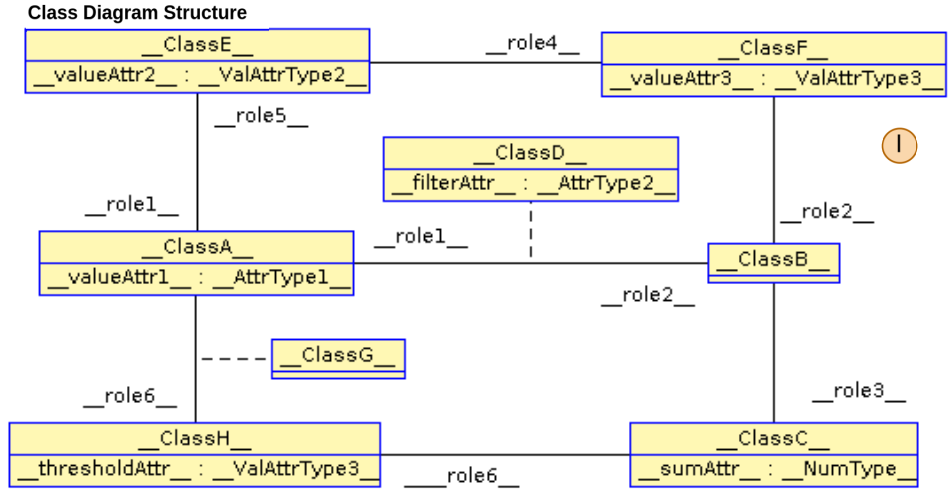
- *OCL template:* Figure 12 (label II) illustrates the OCL template used to generate constraints that express the following restriction: The sum of the values of the attribute `__sumAttr__` over all objects `o` in a collection filtered from the set of `__ClassD__`-objects associated with the current `__ClassA__` instance (`self`) is computed according to the filtering criterion defined by the variable `__filterAttr__`. The partial sums are computed using filtering criteria (`__valuePart1__`, `__valuePart2__`) defined by the attribute `__filterAttr1__`. These are then validated against `__valueAttr1__` to ensure that the overall aggregate equals the combined sums of its filtered subsets, maintaining consistency between the global total and category-based component totals.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 12 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.
- **Example:** We focus on the **CourseMan** model, as shown in Fig. 12, to illustrate this pattern. The underlying constraint ensures that reconciliation by requiring the total passed credits of a student to equal the sum of passed core credits and passed elective credits, guaranteeing consistency between aggregated credit totals and their corresponding category-specific subtotals. The constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv03_ReconcileCoreElectiveBreakdown:
  let total:Integer=self.enrolment->select(e | e.grade >= 'C')
      ->collect(e | e.offering.module.credits)->sum()
  in let core:Integer=self.enrolment
    ->select(e | e.grade>= C' nd e.offering.module.type=CourseType::CORE)
    ->collect(e | e.offering.module.credits)->sum()
  in let elective:Integer=self.enrolment
    ->select(e|e.grade>='C'and e.offering.module.type=CourseType::ELECTIVE)
    ->collect(e | e.offering.module.credits)->sum()
  in total = core + elective
```

We define the specification in DSCL to generate the constraint as follows:

```
@SumConstraint(
    name     = 'courseMan_inv03',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role= {'offering','module'}),
    collect  = {@AttrCond(attr = 'grade', minLim = 'C')},
    sumAttr  = 'credits',
    collect1  = {@AttrCond(attr = 'grade', minLim = 'C'),
        @AttrCond(attr = 'type',  matchVal = 'CORE')},
sumAttr1 = 'credits',
collect2 = {@AttrCond(attr = 'grade', minLim = 'C'),
        @AttrCond(attr = 'type',  matchVal = 'ELECTIVE')},
    sumAttr2  = 'credits',
fixAttr = {@CompCond(val=sumAttr, matchVal=sumAttr1+sumAttr2)}
)
class Student {...}
```

Type03 — Group-Wise/Relative.



OCL Template

```
context __ClassA__ inv __sumConstraint03_invID__:                    II
let v : __NumType__ =  self.__ClassD__
    ->select(c | c.__filterAttr__ = __value__
         and c.__role2__.__role4__.__valueAttr3__ = __value3__)
    ->collect(c | c.__role2__.__role3__.__sumAttr__)->sum()
in  v<= self.__ClassG__.__role6__.__thresholdAttr__
```

Annotation Specification

```
@SumConstraint(                                                      III
  name     = __sumConstraint03_invID__,
  assocCls = @AssocCls(assoc = { __ClassC__, __ClassG__ }),
  rolePath = @RolePath(role = { __role1__,__role2__,__role3__}),
  collect  = {@AttrCond(attr = __filterAttr__, matchVal=__value__),
              @AttrCond(attr =__valueAttr3__, matchVal = __value3__)},
  sumAttr  = __sumAttr__,
  fixAttr  = {@RolePath(role = { __role6__ }),
              @AttrRef(maxLim = __thresholdAttr__)}
 [ifPart   = {@AttrCond(attr =__valueAttr1__, matchVal=__value1__),
              @AttrCond(attr =__valueAttr2__, matchVal=__value2__)}]
)
```

Figure 13: Type 03 of the template specification of the CAP pattern `SumConstraint`.

– *Class diagram structure:* Figure 13 (label I) shows a class diagram depicting three classes, their attributes, and the associations between them. This diagram defines the context of the underlying OCL constraint template.
– *OCL template:* Figure 13 (label II) illustrates the OCL template used to generate constraints that express the following restriction: The sum of the values of the attribute `__sumAttr__` over all objects `o` in a collection filtered from the set of `__ClassD__` objects associated with the current `__ClassA__` instance (`self`) is computed according to the filtering criterion defined by the variable `__filterAttr__` and filtering criterion defined by the valrable `__valueAttr3__`. These are then validated against `__thresholdAttr__`- or `__valueAttr1__`, `__valueAttr1__`, to ensure that the threshold condition is satisfied and that objects do not exceed the permitted limit.

– *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 13 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.
– **Example:** We focus on the domain model, as shown in Fig. 1, to illustrate this pattern. The underlying domain constraint ensures that, within a given term, the total number of credits a student can enroll in does not exceed `maxCreditsPerTerm`. This constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv07_CurTermCreditsLimit:
    let v : Integer = self.enrolment
      ->select(e | e.offering.term.isCurTerm = true
                and e.status = EnrolStatus::ACTIVE)
      ->collect(e | e.offering.module.credits)->sum()
      in v <= self.programEnrolment.program.maxCreditsPerTerm
```

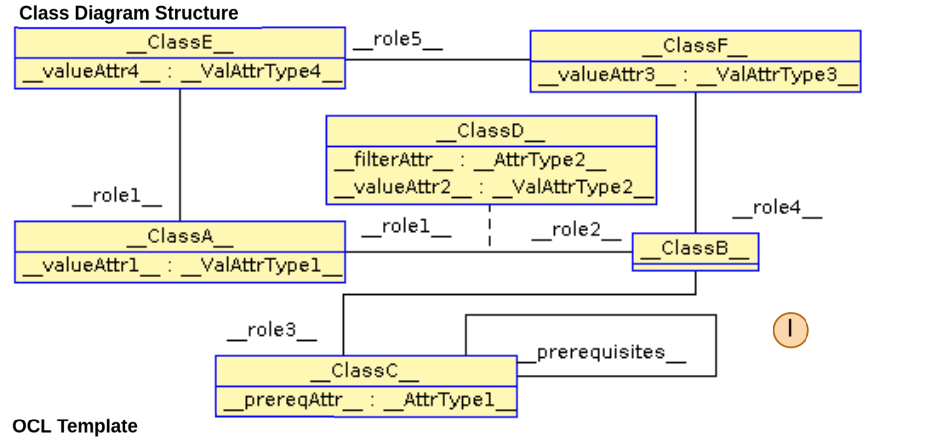
We define the specification in DSCL to generate the constraint as follows:

```
@SumConstraint(
  name     = 'courseMan_inv07',
  assocCls =@AssocCls(assoc ={'Enrolment','programEnrolment'}),
  rolePath =@RolePath(role = {'offering', 'module', 'term'}),
  collect  = { @AttrCond(attr = 'isCurTerm', matchVal = true)
 @AttrCond(attr = 'status', matchVal = 'EnrolStatus::ACTIVE') },
  sumAttr  = 'credits',
  fixAttr  = { @RolePath(role = {'program'}),
           @AttrRef(maxLim = 'maxCreditsPerTerm')}
)
class Student {...}
```

A.2. PrerequisiteConstraint pattern

In this section, we introduce a CAP pattern named PrerequisiteConstraint to illustrate our proposed approach. We then describe the second catalog of CAPs that we have collected and specified from various domain sources.

Type01 - prerequisite/Corequisite/Exclusion Constraints.



OCL Template

```
context ClassA inv __prerequisiteContraint01_invID__:                    II
    self.__ClassD__ -> forAll(e|(e.__filterAttr__= filvalue1__
        and e.__role1.__role4__ = self.__valueAttr1__)
    implies e.__role2__.__role3__.__prerequistes__
       ->forAll(pr |self.__ClassD__
           ->exists(pe |pe.__role2__.__role3__ = pr and
             pe.__filterAttr = __filvalue2__ and
             pe.__valueAttr2__ >= __value2__))
   )
```

Annotation Specification

```
@SumConstraint(                                                          III
  name=__prerequisiteContraint01_invID__,
  assocCls =@AssocCls(assoc={__ClassD__}),
  rolePath = @RolePath(role={__role2__, __role3__}),
  rolePath2=@RolePath(role={__role3__, __role5__})
  collect = {@AttrCond(attr=__filterAttr__,matchVal=__filvalue1__)},
  checkedRole = __prerequisites__,
  ifPart={@AttrCond(attr=__filterAttr, matchStr=__filvalue2__),
         @AttrCond(attr=__valueAttr2__, minLim=__value2__)}
)
```

Figure 14: Type01 template specification of the CAP pattern `PrerequisiteConstraint`.

Pattern Name: PrerequisiteConstraint (Type01 - prerequisite/Corequisite/Exclusion Constraints)

Description: This pattern defines OCL constraints that ensures structural correctness of course relationships and enrollment rules. They require all prerequisites and corequisites to be satisfied (either completed or co-enrolled), forbid self or cyclic dependencies,

and enforce symmetric exclusions. They also demand that appropriate offerings for prerequisite and corequisite modules exist in suitable terms and sequences.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 14 (label I) shows a class diagram depicting the classes, their attributes, and the associations between them, the associations between `__ClassC__` and itself represents collections such as prerequisites, corequisites, or exclusions. This diagram defines the context of the underlying OCL constraint template.
- *OCL template:* Figure 14 (label II) illustrates the OCL template used to generate constraints that express the following restriction:
  For each object of `__ClassA__`, we consider a filtered collection of `__ClassD__`-objects related to self, selected according to the filtering criterion given by `__filterAttr__` (for example, status/term) and the module kind `__prereqAttr__`-(for example, ADVANCED/CORE). For every activated object in this filtered `__ClassD__` collection, and for each prerequisite object reachable through its associated `__ClassC__` instance via `__prerequisites__`, the constraint requires that all such prerequisite objects are satisfied under the defined conditions. In essence, every required related element—whether a prerequisite, a concurrent requirement, or an exclusion under adapted conditions—must be satisfied according to the specified criteria before, or at the moment when, the target element becomes active. Note that the names of the variables in the template begin and end with "___". The variable name `__prerequisiteConstraint01_invID__` implies that this is OCL template type 01 of the PREREQUISITECONSTRAINT pattern.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 14 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model, as shown in Fig. 1, to illustrate this pattern. The underlying constraint ensures that registration is allowed only when all prerequisites have been satisfied. Each prerequisite must meet the minimum required level (for example, $\geq$ 'C'). The constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv20_PrerequisitesSatisfied:
  self.enrolment->forAll(e |
    (e.status=EnrolStatus::ENROLLED
        and e.offering.term = self.curTerm)
    implies e.offering.module.prerequisites->forAll(pr |
        self.enrolment->exists(pe |
          pe.offering.module = pr and
          pe.status = EnrolStatus::COMPLETED and
          pe.grade >= 'C'
        )
      )
  )
```

We define the specification in DSCL to generate the constraint as follows:

```
@PrerequisiteConstraint(
name='courseMan_inv20',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role={'offering', 'module'}),
    rolePath2=@RolePath(role={'term', 'termRecord'})
    collect ={@AttrCond(attr='status',matchVal='EnrolStatus::ENROLLED')},
checkedRole =   'prerequisites',
  ifPart={@AttrCond(attr='status', matchStr='COMPLETED'),
          @AttrCond(attr='grade', minLim='C')}
)
class Student {...}
```

Type02 - Retake and Attempt-Limit Constraints.

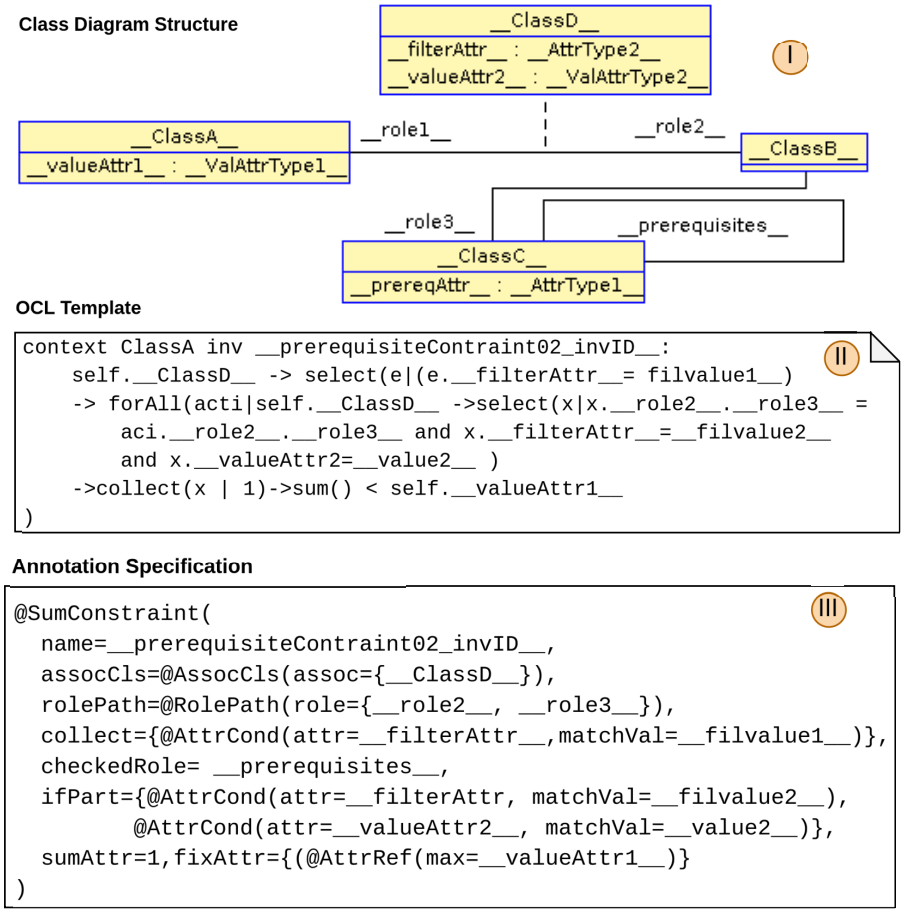


Figure 15: Type02 template specification of the CAP pattern `PrerequisiteConstraint`.

Pattern Name: PrerequisiteConstraint (Type02 – Retake and Attempt-Limit)
Description: This pattern type defines OCL constraints that regulate repetition and failure limits for dependent elements. It ensures that objects are not re-activated

once they have already satisfied the success condition, and that the number of failed attempts for a given related element does not exceed a specified bound. This prevents uncontrolled retries, enforces progression policies, and encourages moving forward instead of repeatedly attempting the same dependent element.

Template: The parameterized OCL expression for this type is defined based on the following main source.

- *Class diagram structure:* Figure 15 (label I) shows a class diagram depicting the context class `__ClassA__`, the association class `__ClassD__`, and the related element class `__ClassC__`, along with their associations. The context object of type `__ClassA__` is linked to multiple `__ClassD__` instances, each referencing a single `__ClassC__` instance and carrying attributes such as status, outcome, or attempt counters. This structure defines the context in which retake and attempt-limit rules are evaluated.
- *OCL template:* Figure 15 (label II) illustrates the OCL template used to generate constraints that express the following restriction:
  For each object of `__ClassA__`, we consider a filtered collection of related `__ClassD__` objects (for example, those currently active or requested), selected according to a filtering criterion defined by `__filterAttr__`. For every activated element in this filtered collection, the template enforces two conditions over all `__ClassD__` objects associated with the same `__ClassA__` instance: (1) No "successful" record exists for the same `__ClassC__` object, where success is determined by `__filterAttr__` and `__valueAttr2__` meeting a given success condition. (2) The number of "failed" records associated with the same `__ClassC__` object, characterized by a failure condition on `__filterAttr__` and `__valueAttr2__`, is strictly less than a configurable upper bound `__valueAttr1__`. Together, these conditions ensure that no new activation is allowed once success has been achieved, and that repeated failures are bounded by the specified limit.
- *Annotation specification:* We extend DCSL with corresponding annotations, as illustrated in Fig. 15 (label III), to capture the parameters of the OCL template. Once these parameters are instantiated, the annotation specification enables automatic generation of the concrete OCL constraint.

Example: We focus on the **CourseMan** model in Fig. 1 to illustrate this pattern. The underlying domain constraint ensures that a student cannot repeatedly fail a module beyond a specified limit. For each currently enrolled module, the number of past failed attempts (for example, completed with grade 'F') must be strictly less than 2. The constraint is expressed in OCL as follows:

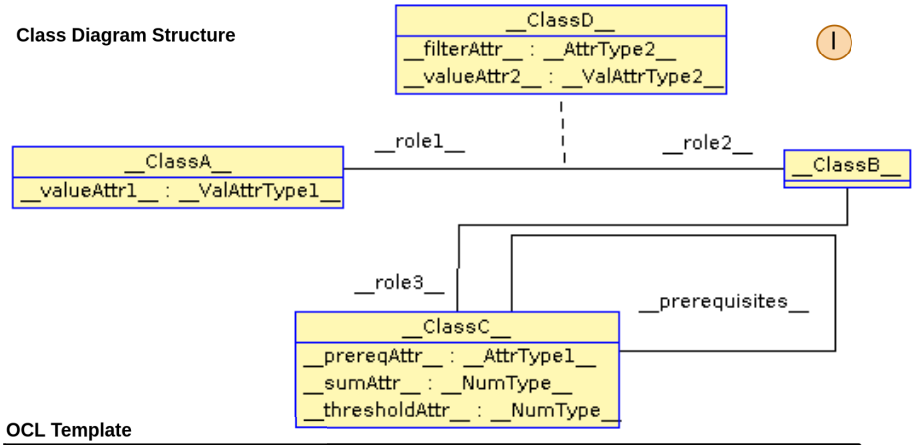
```
context Student
inv courseMan_inv25_MaxFailedAttempts_sum:
  self.enrolment->select(e | e.status = EnrolStatus::ENROLLED)
    ->forAll(activeEnrol |self.enrolment->select(x |
          x.offering.module = activeEnrol.offering.module and
          x.status = EnrolStatus::COMPLETED and x.grade  = 'F')
        ->collect(x | 1)->sum() < 2
    )
```

We define the specification in DSCL to generate the constraint as follows:

```
@PrerequisiteConstraint(
```

```
name='courseMan_inv25',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role={'offering', 'module'}),
    collect ={@AttrCond(attr='status',matchVal='EnrolStatus::ENROLLED')},
checkedRole =   'prerequisites',
  ifPart={@AttrCond(attr='status', matchStr='COMPLETED'),
        @AttrCond(attr='grade', minLim='F')},
    sumAttr=1, fixAttr={(@AttrRef(max=2)}
)
class Student {...}
```

Type03 - Progression / Eligibility Constraints (Credits, GPA)



OCL Template

```
context ClassA inv __prerequisiteContraint03_invID__:
  let v:__NumType__ = self.__ClassD__
    -> select(e|e.__filterAttr__=__filvalue1__ and
         e.__valueAttr2__= __value2__)
    -> colect(e|e.__role2__.__role3__.__sumAttr__) -> sum()
  in self.__ClassD__ ->select(e.__valueAttr2__=__valueAttr3__)
    ->forAll(acti|acti.__role2__.__role3__.__thresholdAttr__>0
  implies v>=acti.__role2__.__role3__.__thresholdAttr__)
```

II

Annotation Specification

```
@SumConstraint(
  name=__prerequisiteContraint03_invID__,
  assocCls=@AssocCls(assoc={__ClassD__}),
  rolePath=@RolePath(role={__role2__, __role3__}),
  collect={@AttrCond(attr=__filterAttr__,matchVal=__filvalue1__),
           @AttrCond(attr=__valueAttr2__, matchVal=__value2__)},
  sumAttr= __sumAttr__,
  checkedRole= __prerequisites__,
  ifPart={@AttrCond(attr=__valueAttr2__, matchVal=__value2__)},
  fixAttr={@RolePath(role = { __role2__,__role3__ })
           @AttrRef(max=__thresholdAttr__)}
)
```

III

Figure 16: Type03 template specification of the CAP pattern `PrerequisiteConstraint`.

Pattern Name: PrerequisiteConstraint (Type03 – Progression / Eligibility)

Description: This pattern type defines OCL constraints that govern progression and eligibility based on accumulated quantitative measures, such as completed workload, performance, or ratings. It enforces that an object may only activate certain related elements (for example, advanced or higher-level ones) if its global measures—such as total completed units or an aggregate score—meet specified threshold values. This guarantees readiness before accessing more demanding dependencies and supports staged progression.

Template: The parameterized OCL expression for this type is defined from the following main source.

- *Class diagram structure:* Figure 16 (label I) shows a class diagram with the context class `__ClassA__`, the association class `__ClassD__`, and the related element class `__ClassC__`. The context class `__ClassA__` may also contain attributes representing aggregate measures, while `__ClassC__` may declare per-element requirements such as `__thresholdAttr__`. This structure defines the context for checking whether global or local thresholds are satisfied before activation.
- *OCL template:* Figure 16 (label II) illustrates the OCL template used to generate constraints that express the following restriction: For each object of `__ClassA__`, a quantitative measure is first computed by aggregating information from its associated `__ClassD__` objects or by reading an aggregate attribute of `__ClassA__`. Next, a filtered collection of `__ClassD__` objects is selected according to a criterion defined by `__filterAttr__` and, optionally, by a classification attribute `__valueAttr__` of the associated `__ClassC__` objects. For each activated `__ClassD__` object in this filtered collection, the template compares the global measure of `__ClassA__` (or an aggregate attribute stored directly on `__ClassA__`) against a requirement associated with the related `__ClassC__` object or a fixed threshold constant. In essence, an element in the filtered collection may only become or remain active if the global measure satisfies the required constraint. This ensures that only objects that have accumulated sufficient prior progress or performance are eligible to participate in the specified relations.
- *Annotation specification:* We extend DCSL with annotations, as shown in Fig. 16 (label III), to capture the parameters of this template, such as the attributes used for filtering, aggregation, and thresholds. Once the parameters are instantiated, the annotation specification can be used to automatically generate the concrete OCL constraint.

Example: We use the **CourseMan** model in Fig. 1 to illustrate this pattern. One underlying constraint requires that a minimum number of accumulated completed credits is reached before certain modules can be taken. Another requires a minimum cumulative GPA before advanced modules can be accessed. For instance, a progression constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv23_MinAccumulatedCreditsForAdvanced:
let totalPassedCredits : Integer = self.enrolment
->select(e | e.status = EnrolStatus::COMPLETED and e.grade >= 'C')
->collect(e | e.offering.module.credits)->sum()
in self.enrolment
->select(e | e.status = EnrolStatus::ACTIVE)
```

```
->forAll(activeEnrol |
activeEnrol.offering.module.requiredCredits > 0 implies
totalPassedCredits >= activeEnrol.offering.module.requiredCredits
)
```

We define the specification in DSCL to generate the constraint as follows:

```
@PrerequisiteConstraint(
name='courseMan_inv23',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role={'offering', 'module'}),
    collect ={@AttrCond(attr='status',matchVal='EnrolStatus::ENROLLED'),
              @AttrCond(attr='grade', minLim='C')},
    sumAttr ='credits',
checkedRole =   'prerequisites',
fixAttr={@RolePath(role={'offering', 'module'},
            @AttrRef(max='requiredCredits')}
)
class Student {...}
```

Type04 - Uniqueness / Anti Double-Booking Constraints

Pattern Name: PrerequisiteConstraint (Type04 – Uniqueness / Anti Double-Booking)
Description: This pattern type defines OCL constraints that enforce uniqueness and prevent conflicting or redundant activations over related elements. It ensures that no two objects in a filtered collection of related elements violate uniqueness or mutual-exclusion policies. This maintains consistency, avoids double-booking, and prevents incompatible dependencies from being simultaneously active.
Template: The parameterized OCL expression for this type is defined as follows.

- *Class diagram structure:* Figure 17 (label I) presents a class diagram involving the context class `__ClassA__`, the association class `__ClassD__`, and the related element class `__ClassC__`. The context object of type `__ClassA__` is linked to many `__ClassD__` instances, each of which refers to one `__ClassC__` instance and may encode status, scope, or time information. Self-associations on `__ClassC__` can express mutual-exclusion or compatibility relations. This structure defines the context for uniqueness and double-booking checks over collections of `__ClassD__` objects.
- *OCL template:* Figure 17 (label II) illustrates the OCL template used to generate constraints that express the following restriction: For each object of `__ClassA__`, a filtered collection of `__ClassD__` objects is selected according to criteria defined by attributes such as `__filterAttr__` (for example, activation status) and some scope attribute. Over this filtered collection, the template enforces that: (1) A chosen projection of each `__ClassD__` object is unique, typically expressed using `isUnique`, thereby preventing multiple active records that represent the same logical entity. (2) No pair of distinct objects in the filtered collection refers, via their associated `__ClassC__` instances, to a pair of elements that are related by a mutual-exclusion or incompatibility relation. This is usually expressed by universally quantifying over pairs of distinct objects and requiring that no forbidden relation holds between their associated `__ClassC__` instances.

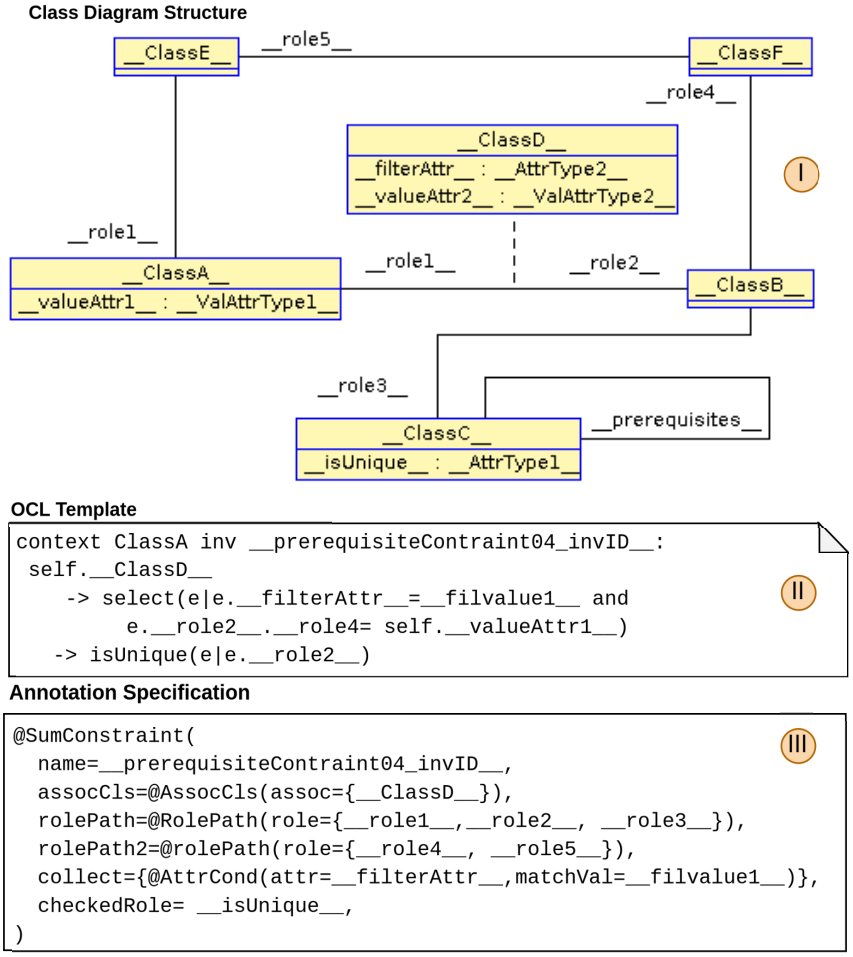


Figure 17: Type04 template specification of the CAP pattern `PrerequisiteConstraint`.

– *Annotation specification:* We extend DCSL with annotations, as illustrated in Fig. 17 (label III), that capture the filtering attributes, projection attributes for uniqueness, and any exclusion relations to be checked. When concrete values are bound to these parameters, the corresponding OCL constraint can be generated automatically from the template.

Example: Using the **CourseMan** model in Fig. 1, one instance of this pattern ensures that, for a given term, a student cannot hold more than one active enrollment for the same offering. The constraint is expressed in OCL as:

```
context Student
  inv courseMan_inv27_NoDuplicateActiveEnrollment:
  self.enrolment ->select(e | e.status = EnrolStatus::ACTIVE
  and e.offering.term = self.curTerm) ->isUnique(e | e.offering)
```

We define the specification in DSCL to generate the constraint as follows:

```
@PrerequisiteConstraint(
  name='courseMan_inv27',
  assocCls=@AssocCls(assoc={'Enrollments'}),
```

```
  rolePath=@RolePath(role={'student,'offering','module'}),
  rolePath2=@RolePath(role={'term','termRecord'}),
  collect={@AttrCond(attr='curTerm',matchVal='true')},
 checkedRole =   'isUnique'
)
class Student {...}
```

A.3. Pattern: ScheduleConstraint

In this section, we introduce a CAP pattern named ScheduleConstraint to illustrate our proposed approach. We then describe the third catalog of CAPs that we have collected and specified from various domain sources. The pattern includes a UML class diagram, together with a set of related OCL constraint templates that govern temporal and logical relationships in enrollment systems through three types:

Type01 - Time Conflict Prevention.

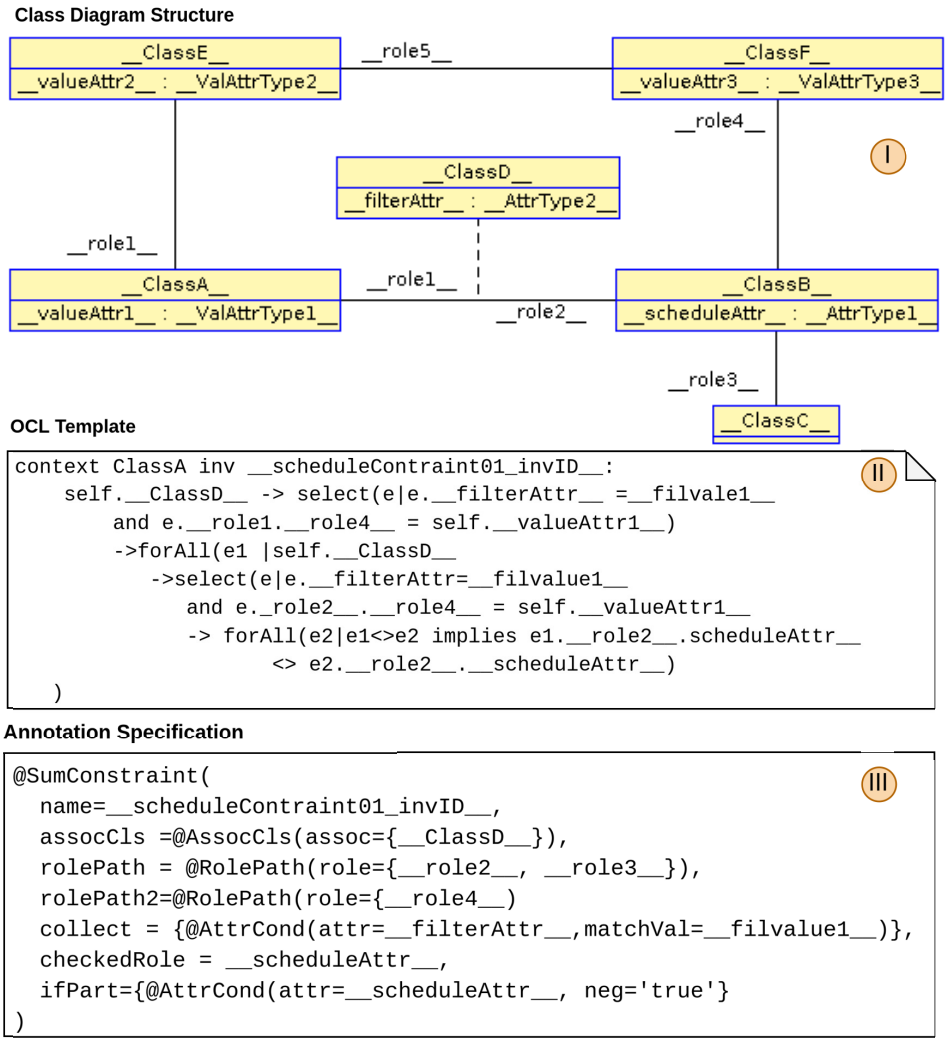


Figure 18: Type01 template specification of the CAP pattern `ScheduleConstraint`.

Pattern Name: ScheduleConstraint (Type01 – Time Conflict Prevention)
Description: This pattern defines OCL constraints for preventing temporal overlaps in scheduling systems. It ensures that no two related objects share the same temporal

slot or schedule identifier within the same scope (term, instructor, room, or student). This prevents double-booking scenarios, guarantees feasible resource allocation, and maintains conflict-free timetables across multiple stakeholders.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 18 (label I) shows a class diagram depicting four main classes: `__ClassA__`, `__ClassD__`, `__ClassB__`, and `__ClassC__`. The diagram includes associations between these classes and attributes for temporal identification. This structure defines the context for conflict detection across related objects.
- *OCL template:* Figure 18 (label II) illustrates the OCL template used to generate constraints that express the following restriction: For each object of `__ClassA__`, we consider a filtered collection of `__ClassD__` objects related to self, selected according to the filtering criterion defined by `__filterAttr__` (such as status, term, or scope). For every object `e1` in this filtered collection, the template requires that no other distinct object `e2` in the same filtered collection shares the same temporal identifier, accessed through the attribute `__scheduleAttr__` (such as `e1.offering.schedule`).

  The constraint enforces that for all pairs of distinct objects in the filtered collection, their schedule projections must be different: `e1.__schedule__ <> e2.__scheduleAttr__`. This guarantees no temporal conflicts exist within the defined scope.

  Note that the names of the variables in the template begin and end with "___". The variable name `__scheduleConstraint01_invID__` implies that this is OCL template type 01 of the ScheduleConstraint pattern.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 18 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model, as shown in Fig. 1, to illustrate this pattern. The underlying constraint ensures that a student cannot enroll in two classes that occur at the same time in the timetable. This prevents double-booking scenarios and guarantees feasible schedules. The constraint is expressed in OCL as follows:

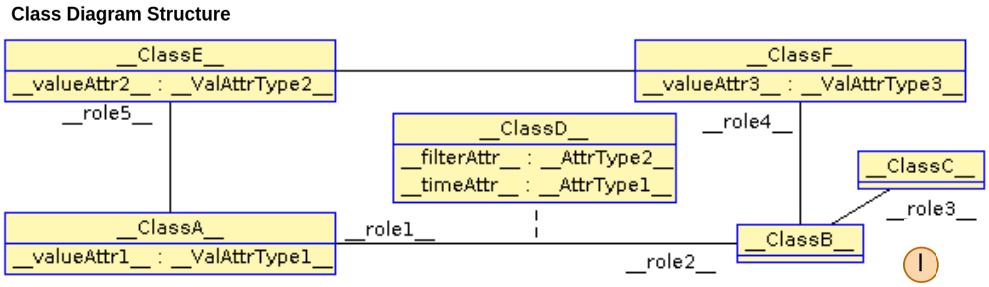
```
context Student
inv courseMan_inv37_NoTimeConflicts:
self.enrolment ->select(e | e.status = EnrolStatus::ACTIVE
and e.offering.term = self.curTerm) ->forAll(e1 |
self.enrolment ->select(e | e.status = EnrolStatus::ACTIVE
and e.offering.term = self.curTerm) ->forAll(e2 |
e1<>e2 implies e1.offering.schedule<>e2.offering.schedule
))
```

We define the specification in DSCL to generate the constraint as follows:

```
@ScheduleConstraint(
    name='courseMan_inv37',
    assocCls=@AssocCls(assoc={'Enrolment'}),
    rolePath=@RolePath(role={'offering'}),
    rolePath2=@RolePath(role={'term','termRecord'}),
```

```
    collect={@AttrCond(attr='status', matchVal='EnrolStatus::ACTIVE')},
    checkRole='schedule',
    ifPart={@AttrCond(attr='schedule', neg='true'
)
class Student {...}
```

Type 2 – Limited Class Switch within Time Window



OCL Template

```
context ClassA inv __scheduleContraint02_invID__:
    self.__ClassD__ -> select(e|e.__filterAttr__ =__filvale1__
        and e.__role1.__role4__ = self.__valueAttr1__)
        and e.__timeAttr__<=__role2__.__role4__.__valueAttr3__)
        ->collect(e | 1) ->sum() <= __threshold__
```

II

Annotation Specification

```
@SumConstraint(
  name=__scheduleContraint02_invID__,
  assocCls =@AssocCls(assoc={__ClassD__}),
  rolePath = @RolePath(role={__role2__)},
  rolePath2=@RolePath(role={__role4__)
  collect = {@AttrCond(attr=__filterAttr__,matchVal=__filvalue1__),
             @AttrCond(attr=__timeAttr__, maxLim=__valueAttr3__)},
  sumAttr=1, fixAttr ={@AttrRef(maxLim = __threshold__)  }
)
```

III

Figure 19: Type01 template specification of the CAP pattern `ScheduleConstraint`.

Pattern Name: ScheduleConstraint (Type02 – Limited Class Switch within Time Window)

Description: This pattern defines OCL constraints for regulating short-term changes or withdrawals within defined temporal boundaries. It ensures that the number of modifications made by an object within a specific time window does not exceed a configured limit. This prevents excessive short-term instability and encourages commitment to enrollment decisions within critical periods.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

– *Class diagram structure:* Figure 19 (label I) shows a class diagram with `__ClassA__`, `__ClassD__`, and temporal attributes such as `__timeAttr__`. The diagram includes

associations and attributes necessary for temporal filtering. This structure defines the context for counting changes within time windows.

- *OCL template:* Figure 19 (label II) illustrates the OCL template used to generate constraints that express the following restriction:
  For each object of `__ClassA__`, we compute the collection of `__ClassD__` objects that occurred within the specified time window, filtered by temporal criteria `__filterAttr__` and status criteria `__timeAttr__`. The template then validates that the size of this filtered collection does not exceed the threshold `__threshold__`. This pattern can be generalized to use sum operations by mapping qualifying objects to 1 and others to 0, then summing the results. ScheduleConstraint pattern.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 19 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model to illustrate this pattern. The underlying constraint ensures that students cannot withdraw from more than 2 classes during the first week of the semester. This prevents excessive early-term instability. The constraint is expressed in OCL as follows:

```
context Student
inv courseMan_inv38_MaxClassSwitchesFirstWeek:
self.Enrolment ->select(e |e.student = self and
    e.offering.term = self.curTerm and
    e.status = EnrolStatus::WITHDRAWN and
    e.withdrawDate <> null and
    e.withdrawDate <= self.curTerm.startDate
)->collect(e | 1) ->sum() <= 2
```

We define the specification in DSCL to generate the constraint as follows:

```
@ScheduleConstraint(
name='courseMan_inv38',
 assocCls=@AssocCls(assoc={'Enrolment'}),
 rolePath=@RolePath(role={'offering'}),
 rolePath2=@RolePath(role={'term','termRecord'}),
 collect={@AttrCond(attr='status', matchVal='EnrolStatus::WITHDRAWN'),
        @AttrCond(attr='withdrawDate', maxLim='startDate')},
 sumAttr=1, fixAttr={ @AttrRef(maxLim = 2)}
)
class Student {...}
```

Type 3 – Maximum Cumulative Class Switch Limit

Pattern Name: ScheduleConstraint (Type03 – Maximum Cumulative Class Switch Limit)

Description: This pattern defines OCL constraints for enforcing global limits on the total number of modifications or withdrawals across extended scopes such as entire terms, academic careers, or programs. It ensures that the cumulative count of changes made by

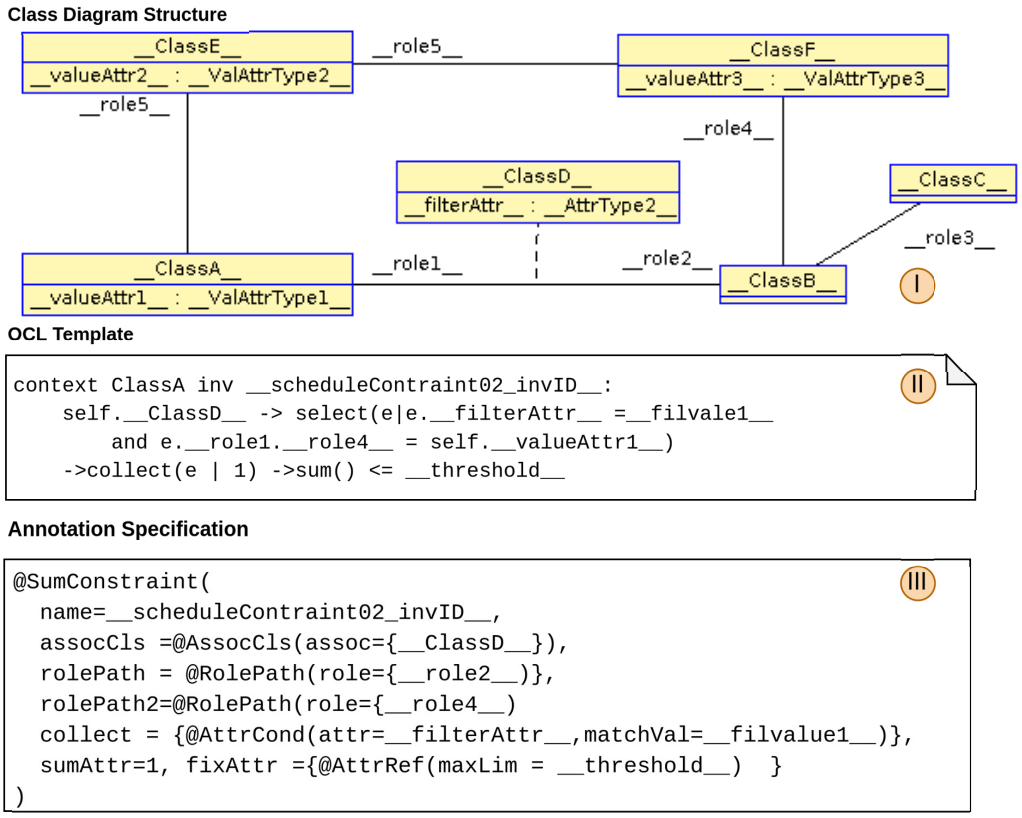


Figure 20: Type01 template specification of the CAP pattern `ScheduleConstraint`.

an object does not exceed configured upper bounds, promoting stability and preventing abuse of enrollment flexibility across broader time horizons.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 20 (label I) shows a class diagram with the context `__ClassA__` relates to `__ClassD__` objects across multiple terms or the entire academic history. The diagram emphasizes cumulative counting without strict temporal boundaries. This structure defines the context for global switch limitation.
- *OCL template:* Figure 20 (label II) illustrates the OCL template used to generate constraints that express the following restriction: For each object of `__ClassA__`, we compute the collection of all `__ClassD__` objects that represent modifications or withdrawals across the specified cumulative scope, filtered by status criteria `_1__` and optionally by broader scope criteria (such as all terms or current program). The template validates that the size (or sum) of this collection does not exceed the global threshold `__threshold__`. This pattern supports both size-based counting and sum-based aggregation for more complex cumulative measures. ScheduleConstraint pattern.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 20 (label III), to capture the parameters of the OCL template. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model to illustrate this pattern. The underlying constraint imposes a hard limit of 5 class withdrawals per semester for any student. This promotes enrollment stability throughout the term. The constraint is expressed in OCL as follows:

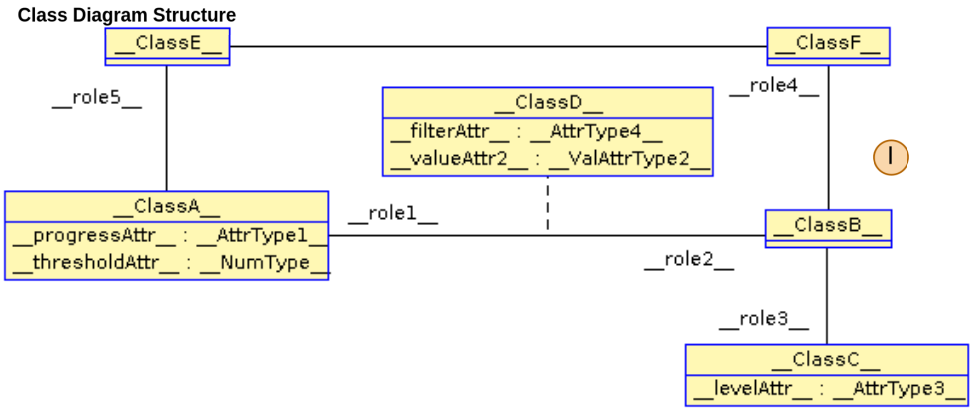
```
context Student
inv cm39_MaxCumulativeClassSwitches:
Enrolment.allInstances()
->select(e |e.student = self and
e.offering.term = self.curTerm and
e.status = EnrolStatus::WITHDRAWN
)->collect(e | 1) ->sum() <= 5
```

We define the specification in DSCL to generate the constraint as follows:

```
@ScheduleConstraint(
name='courseMan_inv39',
 assocCls=@AssocCls(assoc={'Enrolment'}),
 rolePath=@RolePath(role={'offering'}),
 rolePath2=@RolePath(role={'term','termRecord'}),
 collect={@AttrCond(attr='status', matchVal='EnrolStatus::WITHDRAWN')},
 sumAttr=1, fixAttr={ @AttrRef(maxLim = 5)}
)
class Student {...}
```

A.4. Pattern: EligibilityConstraint

In this section, we introduce a CAP pattern named EligibilityConstraint to illustrate our proposed approach. We then describe the fourth catalog of CAPs that we have collected and specified from various domain sources. The pattern includes a UML class diagram, together with a set of related OCL constraint templates that govern qualification criteria for resource access based on actor attributes and history.



OCL Template

II

```
context ClassA inv __eligibilityContraint01_invID__:
    Let v:__AttrType1__ = self.__progressAttr__ in
    self.__ClassD__ -> select(e|e.__filterAttr__ =__filValue__
        and e.__role2.__role3__.__levelAttr = __levelValue__)
        ->forAll(e1 |v >=self.__thresholdAttr__)
```

Annotation Specification

III

```
@SumConstraint(
  name=__eligibilityContraint01_invID__,
  assocCls =@AssocCls(assoc={__ClassD__}),
  rolePath = @RolePath(role={__role2__, __role3__}),
  collect = {@AttrCond(attr=__filterAttr__,matchVal=__filterValue__),
             @AttrCond(attr=__levelAttr__, matVal=__levelValue__)},
  eligiAttr = __progressAttr__,
  fixAttr ={@AttrRef(minLim = __threshold__)}
)
```

Figure 21: Template specification of the CAP pattern `EligibilityConstraint`.

Pattern Name: EligibilityConstraint

Description: This pattern defines OCL constraints that enforce eligibility based on accumulated workload, course level, or progression thresholds. It ensures that an entity may only activate, access, or enroll in certain related elements when quantitative progress measures meet specified minimum requirements. This supports staged progression and prevents premature access to advanced components.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 21 (label I) shows a class diagram depicting the context class `__ClassA__`, the association class `__ClassD__`, and the related class `__ClassC__`. Attributes on these classes capture progression measures (such as `__progressAttr__` for accumulated credits) and requirement thresholds. This structure defines the context of the underlying OCL constraint template.
- *OCL template:* Figure 21 (label II) illustrates the OCL template used to generate constraints that express the following restriction: For each object of `__ClassA__`, a progression measure is obtained either by aggregating information over associated `__ClassD__` objects or by reading an aggregate attribute `__progressAttr__` stored on `__ClassA__`. A filtered collection of `__ClassD__` objects is then selected according to criteria defined by `__filterAttr__` and, optionally, by the classification attribute `__levelAttr__` of the associated `__ClassC__` instances. For every active element in this filtered collection that targets an eligible group of related elements, the template requires that the progression measure satisfies a threshold condition. In essence, an element belonging to a higher level or special category may only become or remain active if the entity has accumulated sufficient prior credits or progression according to the specified requirement. EligibilityConstraint pattern.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 21 (label III), to capture the parameters of the OCL template (for example, navigation paths, filter attributes, progression attributes, and threshold attributes). The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: We focus on the **CourseMan** model, as shown in Fig. 1, to illustrate this pattern. The underlying constraint ensures that a student may only enroll in advanced modules after completing at least a given number of credits. It can be expressed as follows:

```
context Student
inv courseMan_inv55_NoviceAdvancedResourceRestriction:
  let completedCredits:Integer=self.totalCreditsCompleted in
  self.enrolment->select(e|e.status=EnrolStatus::ACTIVE
    and e.offering.module.level = CourseLevel::ADVANCED)
    ->forAll(advEnrol |completedCredits >= 30)
```

A corresponding DSCL specification can be defined using the EligibilityConstraint annotation by binding

```
@PrerequisiteConstraint(
name='courseMan_inv55',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role={'offering', 'module'}),
    collect ={@AttrCond(attr='status',matchVal='EnrolStatus::ACTIVE'),
      @AttrCond(attr='level', matchVal ='CourseLevel::ADVANCED')},
eligiAttr =   'totalCreditsCompleted',
  fixAttr={@AttrRef(minLim=30)}
)
class Student {...}
```

A.5. Pattern: RetakeConstraint

In this section, we introduce a CAP pattern named RetakeConstraint to illustrate our proposed approach. We then describe the fifth catalog of CAPs that we have collected and specified from various domain sources. The pattern includes a UML class diagram, together with a set of related OCL constraint templates that govern the retake conditions for repeating previously attempted resources.

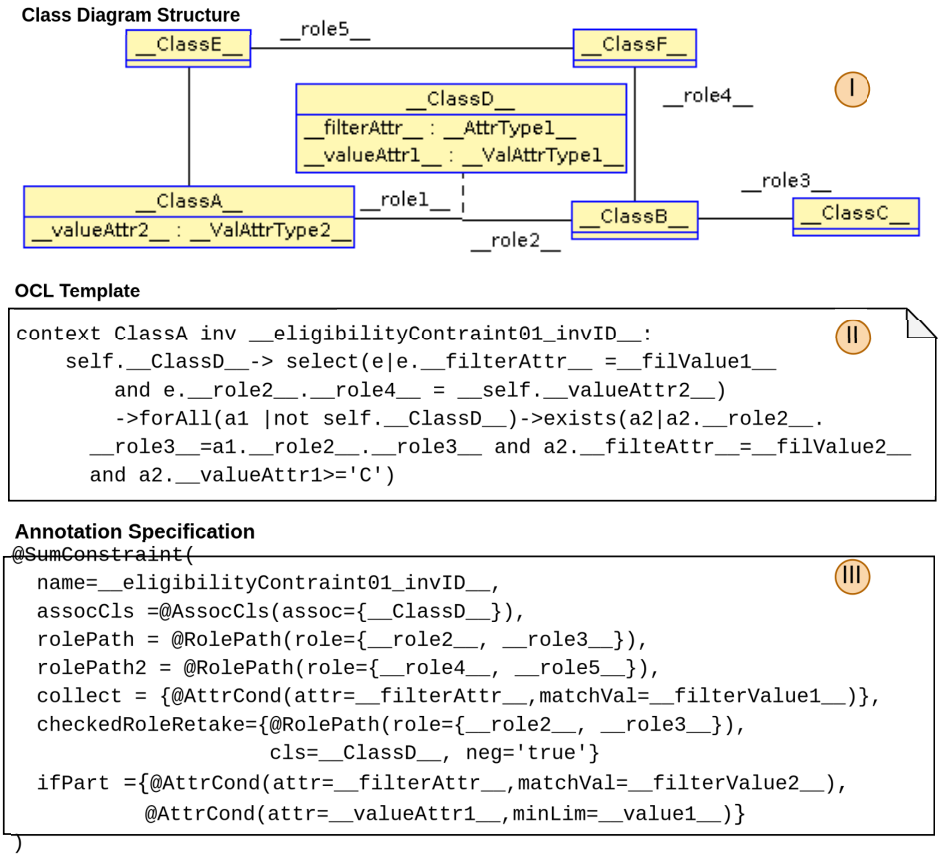


Figure 22: Template specification of the CAP pattern `RetakeConstraint`.

The pattern includes a UML class diagram, together with a set of related OCL constraint templates that govern the retake conditions for repeating previously attempted resources. They establish:

Pattern Name: RetakeConstraint

Description: This pattern defines OCL constraints that forbid activating a new attempt on an element that has already been successfully completed, or that require the existence of at least one prior failed/low-grade attempt before a new attempt is treated as a valid retake. It prevents redundant repetition of already passed elements and ensures that "retake" behavior only applies where a genuine previous failure exists.

Template: The parameterized OCL expression of this pattern is defined based on the following main source.

- *Class diagram structure:* Figure 22 (label I) shows a class diagram depicting the context class `__ClassA__`, the association class `__ClassD__`, and the related element class `__ClassC__`. Attributes on `__ClassD__` include `__filterAttr__` and `__valueAttr__`, which distinguish passed attempts from other outcomes. This structure defines the context for detecting whether a previous passing or failing attempt exists for the same element.
- *OCL template:* Figure 22 (label II) illustrates the OCL template used to generate constraints that express the following restriction:
  For each object of `__ClassA__` (self), we consider a filtered collection of "currently active" `__ClassD__` objects according to criteria `__filterAttr__`. For every such active object `ae` in this collection, the template either: (a) forbids any prior successful attempt on the same `__ClassC__` element by requiring that there does *not* exist a `__ClassD__` object `p` with the same target `__ClassC__` instance, where `p.__filterAttr__` and `p.__valueAttr__` satisfy the success condition; or (b) requires that, if `ae` is considered a retake, then there *must* exist at least one prior attempt `f` on the same `__ClassC__` instance whose status and grade satisfy a failure/low-grade condition. In essence, the template ensures that new active attempts do not redundantly repeat already passed elements, and that retake semantics are only applied when there is an appropriate history of unsuccessful attempts.
- *Annotation specification:* We extend DCSL with new annotations, as illustrated in Fig. 22 (label III), to capture the parameters of the OCL template, such as the navigation from `__ClassA__` to `__ClassD__` and `__ClassC__`, the filter attributes for selecting active attempts, and the status/grade predicates that characterize successful or failed attempts. The annotation specification enables the generation of an OCL constraint once the parameters are assigned concrete values.

Example: In the **CourseMan** model, one instance of this pattern forbids retaking passed courses. It is expressed in OCL as:

```
context Student
inv courseMan_inv76_NoRetakePassedCourses:
  self.enrolment ->select(e | e.status = EnrolStatus::ACTIVE
                    and e.offering.term = self.curTerm)
    ->forAll(activeEnrol|not self.enrolment->exists(passedEnrol|
        passedEnrol.offering.module=activeEnrol.offering.module
        and passedEnrol.status = EnrolStatus::COMPLETED
        and passedEnrol.grade >= 'C')
    )
```

A corresponding DSCL specification can be defined using the RetakeConstraint annotation by binding, and status/grade parameters to the values indicating successful completion.

```
@PrerequisiteConstraint(
name='courseMan_inv25',
    assocCls =@AssocCls(assoc={'Enrolment'}),
    rolePath = @RolePath(role={'offering', 'module'}),
    rolePath2=  @RolePath(role={'term', 'termRecord'}),
    collect ={@AttrCond(attr='status',matchVal='EnrolStatus::ACTIVE')},
    checkedRoleRetake = {@rolePath(role={'offering','module'}),
```

```
                        cls='Enrolment', neg='true'},
    ifPart={@AttrCond(attr='status', matchStr='COMPLETED'),
        @AttrCond(attr='grade', minLim='C')}
)
class Student {...}
```

B. JetBrains MPS: Meta-Programming and Concern Composition

We developed a support tool for UDML that enables the modular separation of concern-specific DSLs, allowing designers to directly model all relevant aspects of the domain.

The tool is organized into four main layers, as illustrated in Figure 23:

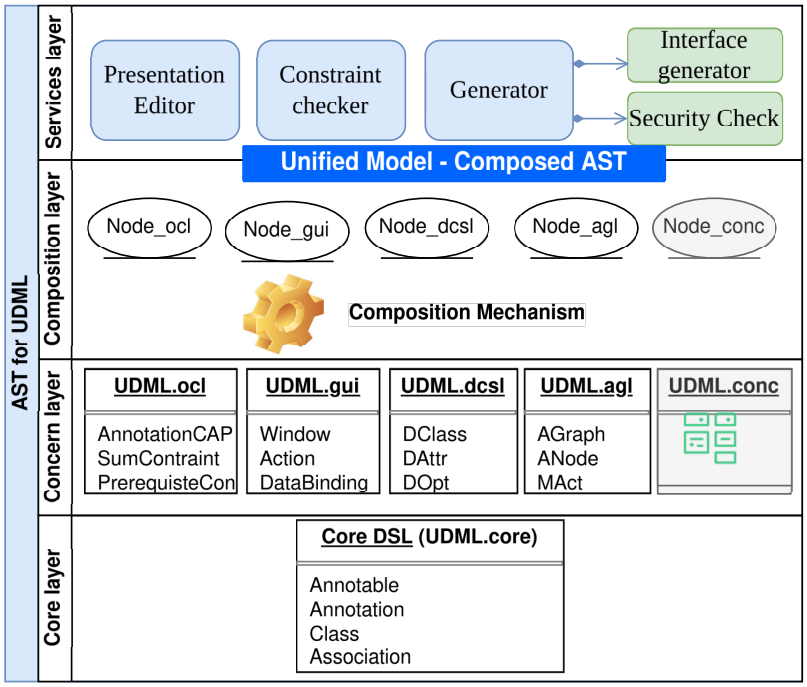


Figure 23: Tool architecture of UDML with the proposed method.

1. The core layer: Defines the syntax of the core DSL (`UDML.core`), providing the foundation for integrating other concerns into UDML. This includes fundamental concepts such as `Annotable`, `Annotation`, `Class`, and `Association`.
2. The concern layer: Defines the AS for each specific concern tailored for development goals. For instance—`UDML.ocl`, `UDML.gui`, `UDML.dcsl`, and the other concerns `UDM.conc`—for general concepts.
3. The composition layer: Facilitates the composition of the defined concerns into the UDML model, utilizing an enhanced composition mechanism to create a unified and consolidated AST.
4. The services layer: Provides graphical interface services that enable designers to perform tasks such as presentation editing, constraint checking, and software/prototype generation, alongside auxiliary services like Security Check.

Our approach is implemented using JetBrains MPS [28, 38], which provides a flexible platform for defining and composing DSLs on the basis of abstract syntax trees (ASTs). Each concern DSL specifies a set of concepts (AST node types), which can be composed and reused across models. Projectional editors enable each concern-specific DSL to present its own dedicated interface, while all contribute to a shared model instance. Integration into existing development workflows is supported through code generation and external

build tools, although it typically requires adaptation rather than seamless incorporation. Each concern DSL is defined as follows:

- **Abstract Syntax**: Defines the structural elements of the language using Concepts, which correspond to nodes in the abstract syntax tree (AST).
- **Concrete Syntax (Editors)**: Specifies the notation or visualization used to display the concepts (e.g., tabular, textual, or symbolic representations).
- **Type System and Constraints**: Ensures model correctness through validity checks and semantic restrictions.
- **Generators**: Transform concern-specific DSL models into target languages, typically embedding them into an object-oriented programming language (OOPL).

The tool facilitates the design, extension, and integration of concern DSLs, validating the approach's expressiveness, feasibility, and satisfiability.

We designed the UDML language as modular packages: `UDML.core` provides the integration foundation, while `UDML.dcsl` capture structural concern, `UDML.agl` capture behavioral concern, and `UDML.ocl` capture the catalog CAP patterns respectively. The central area offers a graphical interface for domain modeling, supported by a palette that enables intuitive construction through rapid insertion of concern-specific elements.

In Figure 24, the left side (label (1)) shows the definition of a template file in the language workbench editor, while the right side (label (2)) presents the resulting `Student` domain model, which is part of the course management system. This domain model is subsequently embedded into the JDA framework to generate the final software artifacts.

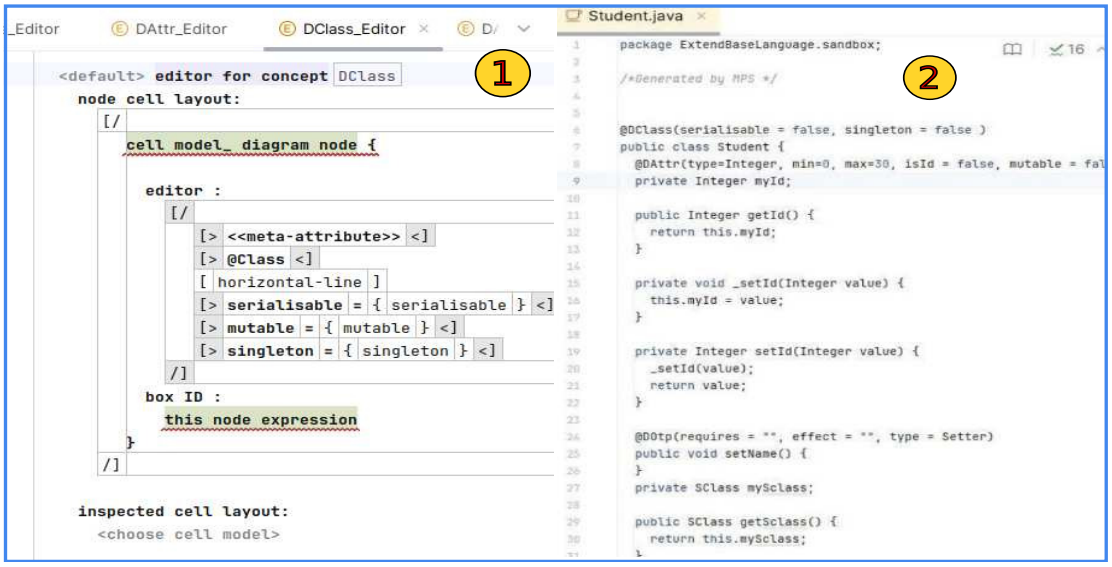


Figure 24: MPS-based Realization and Code Generation via the JDA Framework

C. Evaluating the Expressiveness of CAP

We established 12 evaluation criteria based on the capability of using annotations to specify and enforce domain model aspects. We then evaluated our proposed CAP framework and four representative third-party tools—DCSL, `OpenXAVA`, `Apache Causeway`, and `Actifsource`—using a three-level qualitative scale: (++) fully supported, (+) partially or indirectly supported, and (−) not supported. These twelve criteria are summarized as follows:

1. Support for annotations for quantity constraints: Does the tool allow defining quantity constraints (e.g., a student cannot register for more than 5 courses) through annotations?
2. Support for annotations for logical constraints: Does the tool support defining complex logical constraints (e.g., prerequisites, consequences) through annotations?
3. Support for annotations for derived values: Does the tool allow defining attributes or values calculated from other attributes through annotations?
4. Support for annotations for invariants: Does the tool support defining invariant conditions that must always hold true throughout the object's lifecycle through annotations?
5. Automatic generation of validation code from annotations: Does the tool automatically generate source code to enforce constraints defined through annotations?
6. Support for listening and triggering automatic validation: Does the tool have a mechanism to automatically detect changes and trigger validation based on annotations?
7. Integration capability with object-oriented programming languages (OOPL): Does the tool easily integrate with languages like Java, C#, Python through annotations?
8. Support for complex expressions like forAll, exists: Does the tool allow defining complex logical expressions (e.g., forAll, exists) through annotations?
9. Support for time constraints: Does the tool support defining time-related constraints (e.g., end date must be after start date) through annotations?
10. Customization and extension capability for annotations: Does the tool allow users to define custom annotations to fit specific domains?
11. Integration with development tools (IDE): Does the tool integrate well with integrated development environments (IDEs) to support the use of annotations?
12. Support for multiple programming languages: Can the tool apply annotations to multiple programming languages?

Table 6 presents the comparative results based on the criteria above.

Table 6: Comparing the expressiveness of CAP with DCSL, OpenXAVA, Apache Causeway, and Actifsource.

Criteria	**CAP**	**DCSL**	**OpenXava**	**Apache Causeway**	**Actifsource**
Support for annotations for quantity constraints	(++) @SizeConstraint	(+) @Cardinality	(+) @Size	(+) @CollectionLayout	(++) multiplicity (0..5) in model
Support for annotations for logical constraints	(++) @PrerequisiteConstraint	(+) @DAttr	(+) @AssertTrue	(+) @Property	(++) logical constraints in model
Support for annotations for derived values	(++) @SumConstraint / @ProductConstraint	(+) @DAttr	(+) @Calculation	(+) @Property	(++) derived attributes in model

Criteria	**CAP**	**DCSL**	**OpenXava**	**Apache Causeway**	**Actifsource**
Support for annotations for invariants	(++) @Eligibility-Constraint	(+) @DClass	(+) @Assert-True method check	(+) custom invariant logic	(++) invariants in model
Automatic generation of validation code from annotations	(++) auto-generates from @Sum Constraint	(++) automatic validator generation	(+) generate from @Size	(+) generate layout check	(++) generated model-level constraints
Support for listening and triggering automatic validation	(++) listener triggers validation on change	(++) event-driven validation in runtime model	(−) not supported	(−) not supported	(+) code generation dependent
Integration capability with OOPL	(++) Java annotations (Student class)	(++) Java-based annotation model	(++) JPA annotations	(++) Java domain objects	(++) Java/C++ model code
Support for complex expressions (forAll, exists)	(++) fully supports	(−) not supported	(−) not supported	(−) not supported	(+) defined in model
Support for time constraints	(++) @Time-Constraint	(−) not supported	(+) @Future or @Past	(+) @Property with date check	(++) model-level time rule
Customization and extension capability for annotations	(++) user-defined @Structural Constraint	(−) extendable not supported	(+) @Stereotype extension	(+) @Mixin-extension	(++) user-defined annotation meta-model
Integration with development tools (IDE)	(+) IDE autocomplete for annotations	(++) Eclipse plug-in (DCSL Editor)	(++) Eclipse/ NetBeans integration	(++) IntelliJ/Eclipse integration	(++) Eclipse integration
Support for multiple programming languages	(+) Java, extendable to others	(+) primarily Java	(−) only Java	(−) only Java	(+) Java and C++

D. Required coding level

The criteria related to the amount of manual code developers must write to implement and enforce rules in the domain model, particularly complex constraints:

1. Automation of code generation: How much code does the tool automatically generate? The more automation, the lower the level of manual coding required.
2. Support for complex constraints: Can the tool handle complex constraints (such as OCL expressions) without custom code? Good support for complex constraints reduces the need for manual coding.
3. Integration with domain model: Are the constraints easily integrated into the domain model without much manual adjustment? Tight integration helps reduce additional coding.
4. Learning curve: How much effort does it take for developers to use the tool effectively? A steep learning curve can indirectly increase the level of coding if alternative code needs to be written.
5. Flexibility and customization: Does the tool allow easy customization or addition of custom logic? Good flexibility without requiring much coding is an advantage.

Table 7: The required coding level of CAP

Criterion	**CAP (Constraint Annotation Pattern)**
Automation of Code Generation	Full automation: Automatically generates validation, constraint checking, and OCL enforcement code from annotations (@SumConstraint).
Support for Complex Constraints	Supports complex OCL-style constraints via parameterized templates (SumConstraint). Fully automatable.
Integration with Domain Model	Tight annotation binding with UML-level associations (@RolePath, @AttrCond, @AttrRef), direct mapping to OCL context.
Learning Curve	Moderate – familiar Java syntax, simple annotation logic; minimal domain-specific training needed.
Flexibility and Customization	Sufficient: reusable parameterized patterns.

Table 8: The required coding level of DCSL

Criterion	**DCSL**
Automation of Code Generation	Model-driven generation: Automatically produces class, view, persistence, and controller code from annotated models
Support for Complex Constraints	Manual implementation needed
Integration with Domain Model	All constraints are attached directly to domain classes but lack the capability to integrate multiple domain models
Learning Curve	Steeper – requires learning DCSL grammar and meta-annotation semantics
Flexibility and Customization	Extendable annotation meta-model; supports behavioral and structural extensions

Table 9: The required coding level of **Apache Causeway**

Criterion	Apache Causeway
Automation of Code Generation	Partial automation: Generates CRUD scaffolding, but constraints require manual coding.
Support for Complex Constraints	Limited – requires imperative Java logic for constraint validation.
Integration with Domain Model	Reflection-based, detached from formal domain constraints.
Learning Curve	Easy – based on standard Java + annotations.
Flexibility and Customization	Moderate – extensible via custom Java services.

Table 10: The required coding level of **OpenXava**

Criterion	OpenXava
Automation of Code Generation	Partial: Generates persistence + UI; domain rules coded manually.
Support for Complex Constraints	Basic constraint annotations (@Depends, @Condition); no OCL semantics.
Integration with Domain Model	JPA entity-based integration; rules bound to entities, not model meta-level.
Learning Curve	Easy – low entry barrier with visual modeling.
Flexibility and Customization	Moderate – can add simple validation rules.

Table 11: The required coding level of **Actifsource**

Criterion	Actifsource
Automation of Code Generation	Partial MDD: Generates structural skeleton; logic coded manually.
Support for Complex Constraints	Supports manual rule scripting; no native OCL integration.
Integration with Domain Model	Good meta-model support, but requires manual connector logic.
Learning Curve	High – complex Eclipse-based meta-model setup.
Flexibility and Customization	High – meta-level extension supported.

E. Evaluating the Effectiveness of CAP

The main criteria to evaluate effectiveness. These criteria are designed to reflect the important aspects and main features of DDD:

1. Domain modeling capability: Does the tool effectively support representing concepts and relationships in the business domain?
2. Support for complex constraints: Can the tool efficiently handle complex constraints (such as OCL expressions or advanced business logic)?
3. Consistency between model and code: Is the domain model accurately reflected in the source code, ensuring the 'model is the code' principle of DDD?
4. Effectiveness in automation: Does the tool minimize manual effort through automation (e.g., code generation, interface creation)?
5. Scalability and maintainability: Does the tool support expanding the domain model and maintaining accuracy as the system evolves?

We use three rating levels for the above five criteria: (++) full support or superior performance, (+) sufficient or above-average support, and (−) basic support. Table 12 summarizes the comparison results of CAP with OpenXAVA, Apache Causeway, and Actifsource based on these five criteria.

Table 12: Comparing the effectiveness of CAP to DCSL, OpenXAVA, Apache Causeway, and Actifsource.

Criteria	**CAP**	**DCSL**	**OpenXava**	**Apache Causeway**	**Actifsource**
Domain modeling capability	(+) Direct, rich annotations	(++) domain class, field, associative field, and method	(+) Basic annotations	(+) Basic annotations	(++) Detailed modeling
Support for complex constraints	(++) OCL via annotations	(−) only supporte well-format rule	(−) Basic, requires additional code	(−) Basic, requires additional code	(+) Defined in the model
Consistency between model and code	(+) Annotations embedded in code	(+) software components significantly reduces the model–code gap	(+) Synchronized annotations, but limited	(+) Synchronized annotations, limited	(++) Code generated from the model
Effectiveness in automation	(++) Automatic validation code generation	(++) the quantity and quality of the generated methods	(+) Generates interface, storage	(+) Generates interface, API	(++) Generates entire code
Scalability and maintainability	(+) Adding annotations requires code construction	(+) the quantity and quality of the generated methods	(−) Difficult with complex logic	(−) Difficult with customization	(++) Easy model adjustments

Authors and affiliations

Van-Vinh Le
e-mail: levanvinh@vuted.edu.vn
ORCID: https://orcid.org/0000-0003-3626-7447
(1)Faculty of Information Technology, VNU University of Engineering and Technology,
(2) Vietnam National University, Hanoi
(3) Vinh University of Technology Education

Duc-Hanh Dang
e-mail: hanhdd@vnu.edu.vn
ORCID: https://orcid.org/0000-0003-4564-4080
(1) Faculty of Information Technology, VNU University of Engineering and Technology,
(2) Vietnam National University, Hanoi