International Journal of Software Engineering and Knowledge Engineering

Annotation-Driven Syntax Extensions for OCL Constraint Specification in Domain Models within Domain-Driven Design

Van-Vinh Le[1,2,3] and Duc-Hanh Dang[1,2*]

[1] *Faculty of Information Technology, VNU University of Engineering and Technology*
[2] *Vietnam National University, Hanoi*
[3] *Vinh University of Technology Education*

Received (30 August 2025)
Revised (Day Month Year)
Accepted (Day Month Year)

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.

This paper proposes 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 to capture their semantics, thereby enabling verification on the domain model.

We collected/developed a set of patterns and built a supporting tool for DCSL with CAP, leveraging JetBrains MPS for meta-programming and systematic concern composition, and integrating it with the jDomainApp framework for automated software artifact generation. Evaluation on the CourseMan and OrderMan case studies demonstrated full support for expressiveness and good effectiveness, correctly capturing more than 84% of complex OCL constraints and validating the methodology as a promising solution for practical DDD implementation.

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

1. Introduction

Domain-driven design (DDD) [1] is an approach to developing complex software systems where the primary focus is on the domain and its logic. The core of DDD lies in constructing a domain model that provides the foundation for business knowledge and architectural guidance, with OCL constraints [2] ensuring the precise

*Corresponding author.
Email address: levanvinh@vuted.edu.vn (V.-V. Le), hanhdd@vnu.edu.vn (D.-H. Dang)

specification of business rules. The domain model ensures feasibility, where the code can be implemented and vice versa—allowing the model to be executed and evolved alongside implementation, thereby making it applicable across diverse business contexts. It also ensures satisfiability by capturing and fulfilling all business requirements and critical constraints defined by stakeholders, expressed through a ubiquitous language consistently used by both domain experts and developers in analysis, design, and implementation.

The domain models [3–7] are usually represented in one of the following ways. *First*, UML/OCL class diagrams [8] capture core concepts and their interrelationships. Although they can be translated into executable models or object-oriented programs, a semantic gap remains between model representations and programming languages. Studies [9, 10] have advanced OCL execution and its integration with Java, improving accuracy and verifiability. However, constraint verification still requires compilation, and support for complex OCL remains limited, leaving the full automation of constraint handling an open challenge. *Second*, external DSLs [11–14] introduce custom syntax to express business logic in an accessible way, facilitating constraint specification, automation, and communication between experts and developers. However, they demand substantial effort to build executable toolchains. *Third*, internal DSLs [15–17], especially annotation-based DSLs, embed domain logic within host languages like Java [18], enabling automatic test generation and verifiable implementations. Existing annotation-based approaches [4, 5, 19, 20] 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. *Finally*, recent DSLs such as B-UML [21] demonstrate potential for structural modeling and partial constraint support, but challenges persist in translating UML/OCL into executable code, integrating runtime OCL validation, and ensuring lifecycle correctness.

Our previous works, DCSL [22] and AGL [23], 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 in their ability to capture and execute complex OCL constraints, which are essential for the precise specification of business

rules in DDD [1, 2], and they lack a mechanism or framework to support domain model representation and automatic template generation, thereby preventing a fully executable domain model.

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 uses annotation structures to represent constraint semantics and enable verification directly on the domain model. This approach bridges the gap between the problem domain and the technical space by supporting the automatic generation of software artifacts through reusable CAP. 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 and OrderMan 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 (CAP). Section 5 presents support tools and experiments. Section 6 evaluates and discusses the issues surrounding the method. Section 7 reviews the related work, while Section 8 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 focus is on a course management system, in which the CourseMan problem domain is captured by the domain model shown in Fig. 1. The domain concepts

of this domain correspond to three classes: Student, CourseModule, and SClass. The Student class represents students who register to study in a university. The CourseModule class represents course modules offered by the university. The SClass class represents the student class type that students can choose. Two association classes, SClassRegistration and Enrolments, represent many-to-many associations between the domain classes. The UML class diagram in Fig. 1 captures

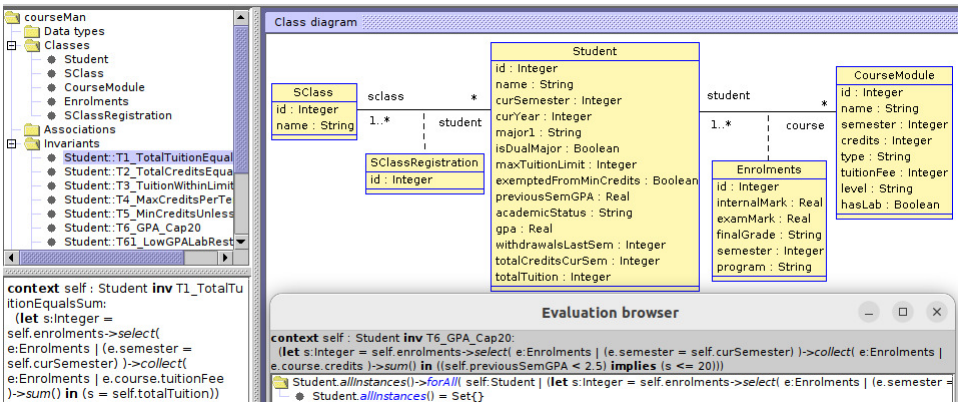

Fig. 1: The **CourseMan** domain model represented as a UML/OCL class diagram.

the structural aspects of the domain model. To specify behavioral and logical aspects that cannot be adequately represented using UML alone, we employ OCL constraints. In particular, the CourseMan domain is refined by introducing an OCL constraint ensuring that the Student entity's displayed tuition fee always equals the sum of tuition fees from all Enrolment records linked to that student in the current semester. This constraint prevents discrepancies between derived totals and the underlying CourseModule data, thereby supporting reconciliation and auditing of tuition information.

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 [22], is to define an annotation-based domain-specific language (DSL), 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 tools 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 [24,25].

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 [14–17] 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, DSLs have been effectively applied to domain modeling and problem-solving in specific contexts, as demonstrated in prior research [11–14, 26]. Collectively, these studies illustrate how the flexibility and adaptability of DSLs can be leveraged to model diverse business domains.

DCSL is an annotation-based domain-specific language (aDSL), introduced in [22], 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.

Table 1: The essential structural constraints (adapted from [22])

Constraints	**Type**	**Descriptions**
Object mutability	Boolean	Whether or not the objects of a class are mutable [27]
Field mutability	Boolean	Whether or not a field is mutable (*i.e.* its value can be changed) [28]
Field optionality	Boolean	Whether or not a field is optional (*i.e.* its value needs not be initialised when an object is created) [28]
Field uniqueness	Boolean	Whether or not the values of a field are unique [28]
Id field	Boolean	Whether or not a field is an object id field [28]
Auto field	Boolean	Whether or not the values of a field are automatically generated (by the system) [28]
Field length	Non-Boolean	The maximum length (if applicable) of a field (*i.e.* the field's values must not exceed this length) [28]
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) [28]
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) [28]
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 defined by a metamodel whose meta-concepts consist of core object-oriented programming language (OOPL) constructs and constraint-related elements. Specifically, the core meta-concepts include: `Domain Class`, `Domain Field`, `Associative Field`, and `Domain Method`. The remaining meta-concepts, along with the properties of all concepts, are defined to express essential OCL constraints, which are summarized in Table 1.

Fig. 2 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,

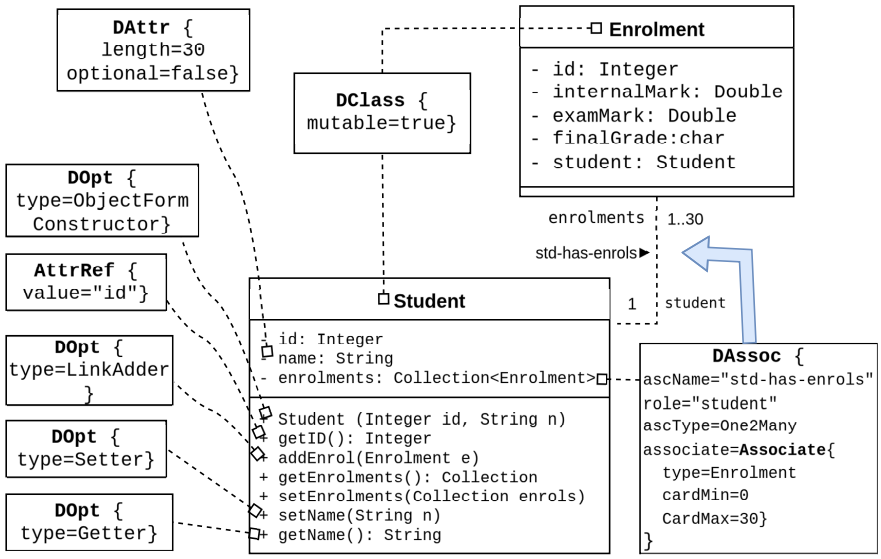

Fig. 2: A simplified representation of the CourseMan domain model using DCSL.

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

This section presents the methodology annotation-driven syntax extensions for OCL constraint specification within DDD.

The approach enables a systematic transformation of domain requirements into executable software prototypes through unified domain modeling, as depicted in Fig. 3. By leveraging annotation-based syntax extensions, it tightly integrates structural and behavioral semantics, maintaining continuous alignment between domain understanding and software implementation. The methodology proceeds in two phases:

First, define domain patterns. Domain modelers extract requirements and distill them into reusable patterns. Each pattern (a **Constraint-Annotation Pattern** (**CAP**)) encapsulates:

- *UML class diagram*: core classes, attributes, associations, and generalizations.
- *OCL specification*: business rules and invariants that ensure logical consistency.
- *Annotations*: mappings from OCL to DCSL/aDSL constructs that attach semantics to model elements.

Second, iterative application of patterns.

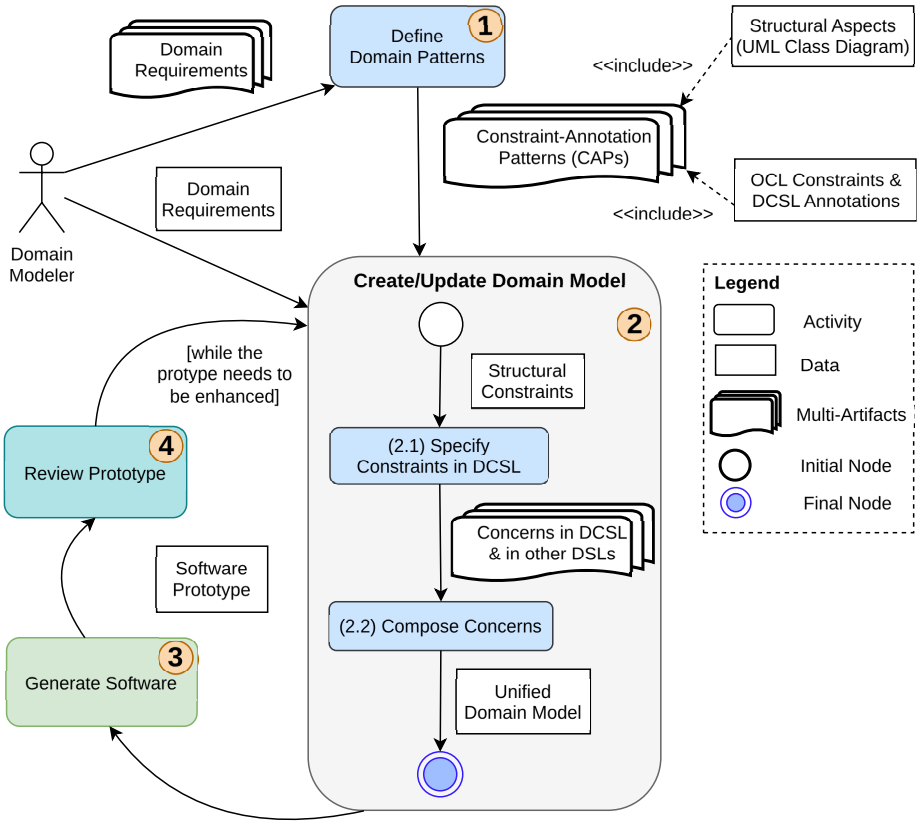

Fig. 3: Overview of our approach.

(i) Create/update the domain model. Modelers instantiate and refine the domain model using the defined CAP; structural and behavioral constraints are expressed explicitly in DCSL, yielding a set of concerns across DCSL and related DSLs.
(ii) Concern composition. The concerns are composed into a unified domain model, integrating structural and behavioral aspects into a coherent semantic artifact.
(iii) Software prototype generation. The unified domain model serves as the blueprint for automatic prototype generation using frameworks such as jDomainApp. The prototype is delivered to stakeholders for evaluation.
(iv) Iterative refinement. Stakeholder feedback drives updates to requirements, improvements to the CAP, and refinements to the domain model. The cycle of model update, artifact generation, and review repeats until functional and quality criteria are satisfied.

4. Extending OCL constraints in domain models via annotation-driven syntax

Our method focuses on addressing the proposed challenges. This section explains our constraint annotation patterns-based approach for representing class diagrams and OCL constraint specification, and for mapping them to annotations within the domain model, integrated into DCSL via the CAP mechanism. Each UML class diagram, along with a set of related OCL constraint templates that represent similar business rules, forms a template model. This template model is then mapped to an annotation template. We perform this translation by applying template domain models, which are defined to correspond to specific problem domains.

4.1. *Specifying constraint annotation pattern*

We are particularly interested in the design of the pattern forms [29–31]. To keep the patterns general, for each pattern form, we present a UML class diagram along with a generalized OCL constraint specification and an annotation template, unified with the domain model that realizes it. The constraint annotation pattern form is a *parameterized* configuration of an OCL constraint specification and annotation template, in which elements of the structure, the OCL constraint specification, and the annotation template are named according to the generic roles they play.

We present a formal, CAP-driven domain modeling method that applies a mathematical and systematic approach, using the catalog of CAPs to map modeling elements and automatically generate domain class specifications. This process ensures that OCL constraint specifications are applied accurately and consistently in the generated source code. A CAP consists of four main components:

(1) Pattern name: The name identifies a classified, reusable pattern representing distinct business rules and domain-specific constraints. It reflects the core concept or recurring structure of a problem within a specific domain (*e.g.* `SumConstraint`, `ScheduleConstraint`, etc.).
(2) Description: The symbols and notations used in the class diagram, OCL constraints specification, and annotations. A clear explanation of the problem the pattern addresses, including its relevance and context within the domain.
(3) Template: The template defines the syntactic structure of the pattern and includes:
    (a) Class diagram: UML representation of classes, attributes, associations, and generalizations that form the structural basis of the pattern.
    (b) OCL constraints specification: Formal expressions that define business rules, constraints to ensure model consistency.
    (c) Annotations: Annotation serves as a semantic bridge between high-level formal constraint specifications and concrete implementation in aDSLs or OOPLs. Each annotation is defined

through a mapping from the corresponding OCL expressions and is structurally bound to model elements—such as classes, attributes, or associations—in the class diagram.

(4) Semantics: This component specifies the meaning and interpretation of the pattern: Describes how the elements of the pattern behave or interact during runtime; Explains the logical implications of the constraints and structure; Provides guidelines for correct usage, transformation rules, or mapping to OCL to annotation.

For the problem domains of CourseMan and OrderMan, the systems incorporate numerous business rules and constraints in diverse forms. To enhance expressiveness and representational power within the domain model, we conducted a survey and classified these rules and constraints into a catalog of CAPs, as follows:

(1) Well-formedness constraints - `EssentialConstraint`: Rules ensuring that the model or data complies with valid formats and structures.
(2) Quantitative limits – `SumConstraint`: Restrictions on quantities, amounts, or minimum/maximum values.
(3) Dependency and prerequisite constraints – `PrerequisiteConstraint`: Conditions based on dependency relationships or prior requirements.
(4) Scheduling and conflict constraints – `ScheduleConstraint`: Rules related to scheduling, resource allocation, and conflict resolution.
(5) Entity qualification constraints – `EligibilityConstraint`: Conditions determining whether an entity or actor is eligible to participate in an activity or process.
(6) Retry or reattempt rules – `RetakeConstraint`: Regulations governing repeated or retried actions after an unsuccessful attempt.
(7) Capacity constraints – `SizeConstraint`: Limits on capacity or the number of entities within a given scope.
(8) Time and deadline constraints – `TimeConstraint`: Timeframes, deadlines, or valid periods for performing an action.
(9) Computation and derivation rules – `SumProduct`: Rules for calculating or deriving values from existing data.
(10) Status-based constraints – `StatusConstraint`: Rules depending on the status or condition of an entity (*e.g.* scholarship, debt, suspension).
(11) Organizational structure constraints – `StructuralConstraint`: Constraints related to organizational structures or hierarchical arrangements.

Each pattern includes a variant of the OCL template integrated with the domain model to represent different business rules. These patterns are encapsulated within a new framework that defines a mechanism for creating a fully executable unified domain model. This framework provides a structured approach for defining, applying, and enforcing domain constraints consistently throughout the software development lifecycle.

The `EssentialConstraint` of CAP was introduced in DCSL [22] to support domain model design. In this section, we provide complete explanations for all ten CAP patterns, each representing a different constraint annotation pattern, we focus on the `SumConstraint` pattern and provide futher details in **??**.

4.2. *SumConstraint pattern form*

Pattern name: SumConstraint.

Generic description for the SumConstraint pattern.

`SumConstraint` is an important constraint pattern in domain modeling, designed to control and manage quantities, values, or resources through aggregation operations. This pattern ensures that the sum of values in collections or relationships complies with domain-specific rules and constraints. It governs resource allocation and value aggregation based on various conditions, including types such as:

Type 1 – Total value calculation constraint: Ensures that the aggregate value stored on an entity always equals the total of its related components. Used to prevent discrepancies between derived data and source data, supporting reconciliation and auditing.

Example: Fig. 4(A) depicts the UML class diagram of CourseMan with an OCL constraint stating that the displayed tuition fee for a student must equal the sum of the tuition fees of all courses registered in the current semester, represented through specification annotations that manage enrollment. Fig. 4(B) illustrates the resulting `SumConstraint` CAP pattern.

The OCL constraint: a student's displayed tuition fee equals the sum of fees for all registered courses in the current semester

```
context Student inv T1_TotalTuitionEqualsSum:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.tuitionFee)->sum()
  in s = self.totalTuition
```

The corresponding annotation is mapped to its feature

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "tuitionFee", minAttr = "totalTuition",
  maxAttr = "totalTuition"
  )
```

Type 2 – Quantity aggregation constraint: Constrains derived quantities (amounts, weights, scores) to equal the sum of filtered constituent elements. Supports partitioning by status, type, or period, ensuring consistency between reports and detail-level data.

Example: The total number of credits currently displayed must exactly match the sum of credits for all courses in the current semester, as depicted in Fig 1 by CourseMan.

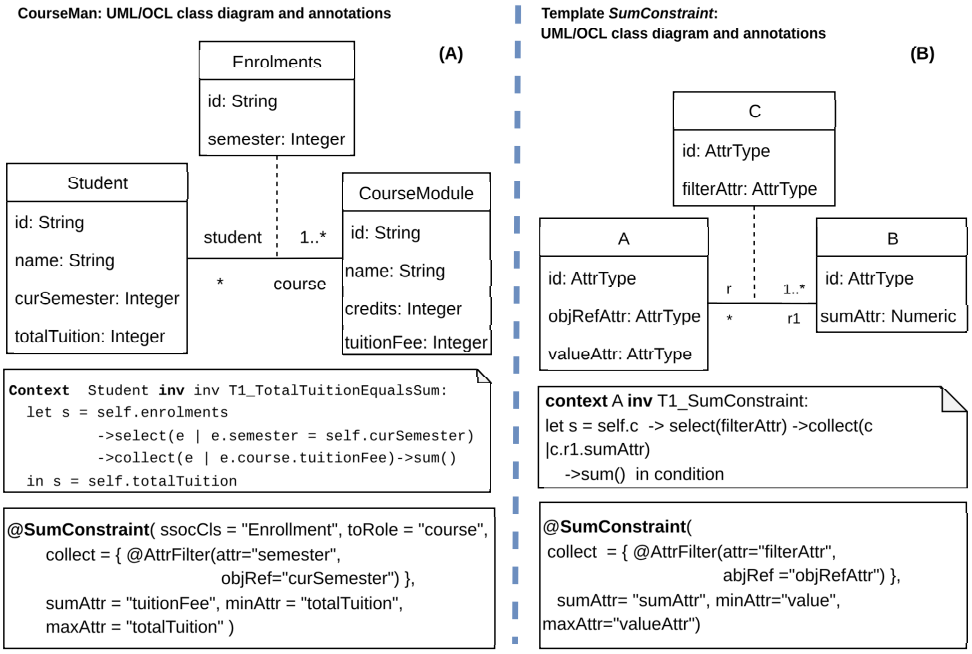

Fig. 4: (A, Left) the annotated UML/OCL class diagram of a CourseMan variant handling enrollment totals and tuition fees. (B, Right) the **SumConstraint** CAP pattern, shown as an annotated UML/OCL class diagram.

```
context Student inv T2_TotalCreditsEqualsSum:
  self.totalCreditsCurSem = self.enrolments
        ->select(e | e.semester = self.curSemester)
        ->collect(e | e.course.credits)->sum()
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", minAttr = "totalCreditsCurSem",
  maxAttr = "totalCreditsCurSem"
  )
```

Type 3 – Financial limit enforcement: Limits cumulative monetary amounts from exceeding predefined thresholds (either attribute-based or constant). Supports dynamic thresholds based on personal profiles, rolling time windows, and controlled exceptions to prevent overspending.

Example: The total tuition fee for the current semester must not exceed the student's individual limit.

```
context Student inv T3_TuitionWithinLimit:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.tuitionFee)->sum()
  in s <= self.maxTuitionLimit
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "tuitionFee", maxAttr = "maxTuitionLimit" )
```

Type 4 – Maximum resource allocation constraint: Caps the allocation of resources (credits, slots, seats) within a given context or period. Often combined with activation conditions by role, year, or target group. Useful for ensuring fairness, limiting abuse, and preserving system capacity.

Example: The total number of credits per semester must not exceed 30.

```
context Student inv T4_MaxCreditsPerTerm:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.credits)->sum()
  in s <= 30
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", maxValue = 30 )
```

Type 5 – Minimum resource requirement constraint: Sets a floor for usage or accumulation to ensure minimum participation, with exemption options for special cases (orCondition). Commonly used for eligibility criteria, pacing progress, or compliance requirements.

Example: Students must complete a minimum of 15 credits in the semester, unless an exemption flag is set.

```
context Student inv T5_MinCreditsUnlessExempt:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.credits)->sum()
  in (not self.exemptedFromMinCredits) implies s >= 15
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", minValue = 15,
  orCondition = "exemptedFromMinCredits" )
```

Type 6 – Performance-based resource limitation: Adjusts caps based on previous performance indicators (scores, error rates). When metrics fall below thresholds, stricter limits are enforced in the subsequent period. Encourages quality, mitigates risk, while maintaining transparency through clear quantitative criteria.

Example: When the GPA of the previous semester is less than 2.5, the total number of credits for the current semester must be less than or equal to 20.

```
context Student
inv T6_GPA_Cap20:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.credits)->sum()
  in self.previousSemGPA < 2.5 implies s <= 20
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", maxValue = 20,
  ifPart = @AttrConstr(attr="previousSemGPA", up=2.5) // attr < 2.5
)
```

Type 7 – Conditional resource restriction: Applies conditional limits based on status/labels/qualifiers. Limits are applied to filtered subsets (type, level, category).

Example: When in PROBATION status, the number of ADVANCED courses in the semester must be 2 (counting elements).

```
context Student inv T7_ProbationAdvancedLimit:
    let sum = self.enrolments->select(e | e.course.level = 'ADVANCED'
    and e.semester = self.curSemester) ->collect(e | 1)->sum()
    in self.academicStatus = 'PROBATION' implies sum <= 2
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = {
               @AttrFilter(attr="semester", objRef="curSemester"),
               @AttrFilter(attr="course.level", objValue="ADVANCED")},
  sumAttr = "1", maxValue = 2,
  ifPart = @AttrConstr(attr="academicStatus", objValue="PROBATION")
  )
```

Type 8 – Penalty-based future limitation: Imposes future limits based on past violations (cancellations, withdrawals, debts). When violations exceed a threshold, allocation caps are reduced in the next period.

Example: If the number of course withdrawals in the previous semester is greater than 2, the total credits for the current semester must be less than or equal to 12.

```
context Student inv T8_WithdrawalPenalty:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.credits)->sum()
  in self.withdrawalsLastSem > 2 implies s <= 12
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", maxValue = 12,
  ifPart = @AttrConstr(attr="withdrawalsLastSem", low=2) // attr > 2
  )
```

Type 9 – Multi-category balanced distribution: Ensures balanced allocation across multiple categories (majors, programs, portfolios) by applying minimums/maximums for each group.

Example: Dual-degree students must complete ¿=12 credits for each major in the semester (illustrated for major1; apply similarly for major2).

```
context Student inv T9_MinPerMajor1:
  let s = self.enrolments -> select(e | e.semester = self.curSemester
  and e.program = self.major1) -> collect(e | e.course.credits)->sum()
  in self.isDualMajor implies s >= 12
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = {@AttrFilter(attr="semester", objRef="curSemester"),
             @AttrFilter(attr="program", objRef="major1")},
  sumAttr = "credits", minValue = 12,
  ifPart = @AttrConstr(attr="isDualMajor", objValue="true")
  )
```

Type 10 – Structured allocation framework: An integrated framework that combines multiple SumConstraints (min, max, conditional, group-based) into a comprehensive policy. Defines evaluation order, priority strategies, and conflict resolution.

Example: A combined min&max policy on the same criterion: First-year students must register for between 12 and 18 credits in the current semester.

```
context Student inv T10_Year1_12to18:
  let s = self.enrolments->select(e | e.semester = self.curSemester)
          ->collect(e | e.course.credits)->sum()
  in self.curYear = 1 implies (s >= 12 and s <= 18)
```

The corresponding annotation is mapped to its feature:

```
@SumConstraint(
  assocCls = "Enrollment", toRole = "course",
  collect = { @AttrFilter(attr="semester", objRef="curSemester") },
  sumAttr = "credits", minValue = 12, maxValue = 18,
  ifPart = @AttrConstr(attr="curYear", eq=1)
  )
```

Template SumConstraint.

The left side of Fig. 5 shows the UML class diagram of the pattern. This pattern consists of three classes: A (source class), B (target class), and C. Class C is the association class with associations between A and B. The associations are named r and r2, representing the relationship between class A and class B (*e.g.* , student, course). The right side of Fig. 5 shows a collection of OCL constraints involving quantity-based rules, exact summation, and value accumulation constraints on time-based activity domains, enabling complex constraint-checking systems to dynamically regulate operations between A, B, and C. The bottom right of Fig. 5

shows the corresponding annotation template, referred to as the `@SumConstraint` pattern.

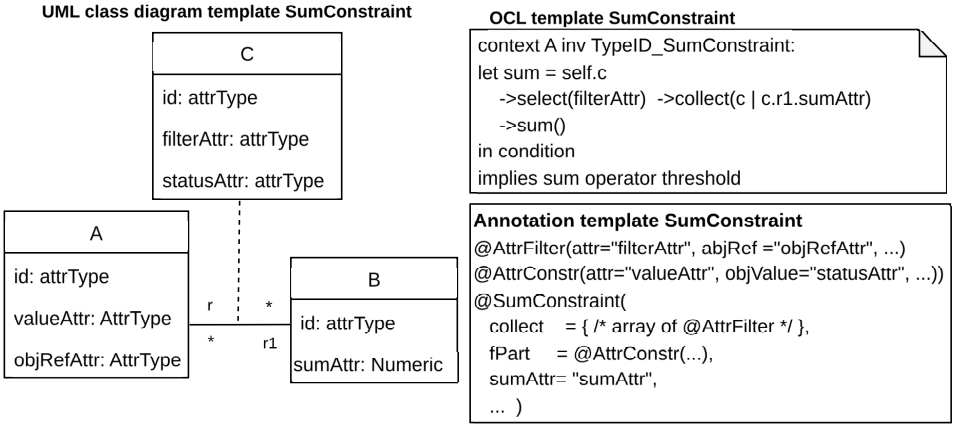

Fig. 5: Template domain model **SumConstraint**.

Semantics: SumConstraint.

The `SumConstraint` pattern regulates quantity-based constraints over a set of objects related via two levels of associations. It enables summation, accumulation, and conditional enforcement of domain rules over time-based or attribute-filtered contexts.

The general OCL template for `SumConstraint` is represented as follows:

```
context A inv TypeID_SumConstraint:
    let sum = self.c ->select(filterCondition)
        ->collect(e | e.r1.sumAttribute) ->sum()
        in condition implies sum operator threshold
```

The general annotation template for `SumConstraint` is represented as follows:

1) @AttrFilter (Filter on associated elements)

```
@AttrFilter(
  attr = "", // dot-path on the associated element
  value = , // (optional, numeric) numeric comparison
  objValue = "", // (optional, string/enum/bool) object comparison
  objRef = "", // (optional) context reference for comparison
  boolAttr = "" // (optional) boolean dot-path;
                  //if provided without objValue -> interpreted as attr == true
)
```

2) @AttrConstr (Activation condition on context)

```
@AttrConstr(
  attr = "", // dot-path on 'self' (the context)
  value = , // alias for 'eq' (for backward compatibility)
  eq = , // numeric equality (e.g.: eq = 1); use objValue for string/enum/bool
  up = , // means '<' (strict upper bound)  e.g.: up = 2.5 -> attr < 2.5
  upEq = , // means '<=' (upper inclusive)
```

```
  low = , // means '>' (strict lower bound)
  lowEq = , // means '>=' (lower inclusive)
  objValue = "" // comparison with string/enum/bool: attr == objValue
)
```

3) SumConstraint

```
@SumConstraint(
  assocCls = "", // name of the associated class
  toRole = "", // role to navigate from 'e' to the attribute being summed
  collect = {array of @AttrFilter(...)}, // filter applied to elements 'e'
  sumAttr = "", // attribute to be summed
  minValue = , // (optional) numeric lower bound
  maxValue = , // (optional) numeric upper bound
  minAttr = "", // (optional) dot-path on 'self', e.g.: "minCreditsRequired"
  maxAttr = "", // (optional) dot-path on 'self', e.g.: "maxTuitionLimit"
  ifPart= @AttrConstr(...),//(optional) constraint applies IF 'ifPart' is satisfied
  orCondition= "" // (optional) boolean dot-path on 'self'
)
```

Based on a survey of CourseMan's business rules and common constraint types in learning management systems, UML/OCL [2,8] is used for model validation and to check the correctness of the UML/OCL model [32–35]. However, to address all constraints—such as data integrity constraints, history and tracking constraints, reporting constraints, and others—we separate and discuss them in detail in **??**.

5. Tool support

We have implemented this methodology using JetBrains MPS, exploiting its projectional editing capabilities [36] to support the entire workflow from CAP definition to executable model generation. The tool integrates:

- **Graphical pattern definition**: Intuitive drag-and-drop interfaces for constructing CAP with their structural, constraint, and annotation components.
- **Concern-based composition**: Systematic integration of structural and behavioral aspects into coherent unified models.
- **Executable model generation**: Automatic production of artifacts deployable in the JDomainApp (JDA) framework.

As illustrated in Fig. 6, the tool architecture comprises three components:

(1) Meta-concept definition: Each CAP is formalized as a meta-concept within the UDML metamodel.

(2) Interactive design interface: A modeling workspace combining graphical, textual, and tabular notations for seamless domain model creation.

(3) Pattern-based modeling support: Modular CAP with contextual submenus for quick application of domain-specific business rules.

The complete UDML framework [37] encapsulates these components, enabling generation of executable unified domain models that integrate directly into the JDA framework.

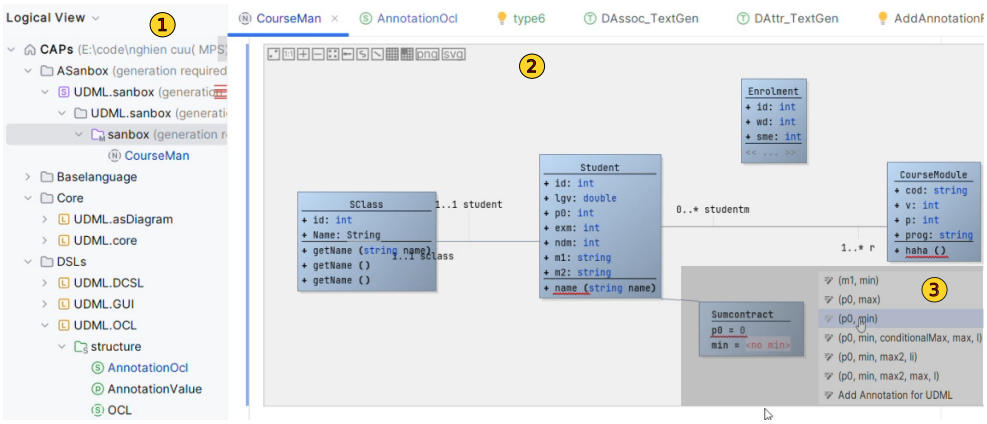

Fig. 6: Supporting tool developed based on JetBrains MPS.

We define a mechanism to execute the unified domain model—comprising CAP integrated into UDML—within an OOPL, ensuring that the model is both executable and deployable in real-world applications. The execution pipeline incorporates on-the-fly OCL evaluation, implementing a runtime verification system [38] that continuously checks the unified model's compliance with the formal constraints embedded in the CAP. Violations are detected immediately, preventing inconsistencies and preserving semantic integrity. This verification mechanism supports compositional execution of the unified domain model and operates through three stages: *Listen* — capture runtime events from executing components, *Process* — evaluate affected model elements against OCL constraints, and *Validate* — confirm compliance or initiate corrective actions. The implementation, illustrated in Fig. 1, is realized using Spring Boot and the JDA framework [39]. From the unified domain model, developers generate: (i)the CAP-integrated UDML model — defining all domain components and classes, and (ii) the framework specifications — ensuring consistent constraint enforcement at runtime. The resulting Java program embeds domain-specific execution semantics, dynamic constraint monitoring, and full alignment with the modeled CAP, thus completing the modeling–execution–verification pipeline.

It is important to note that the DCSL, CAP domain model, and UDML specifications are all encoded in Java, as depicted in the middle and right panels, respectively, of Fig. 7. The left panel of Fig. 7 displays a list of module classes and their corresponding domain classes, each annotated with OCL-mapped annotations from the CAP specification. The right panel presents an annotation definition representing a `SumConstraint`.

To develop the CourseMan software with a GUI, as demonstrated in Fig. 8, the unified DCSL model and CAP domain model are combined into an executable

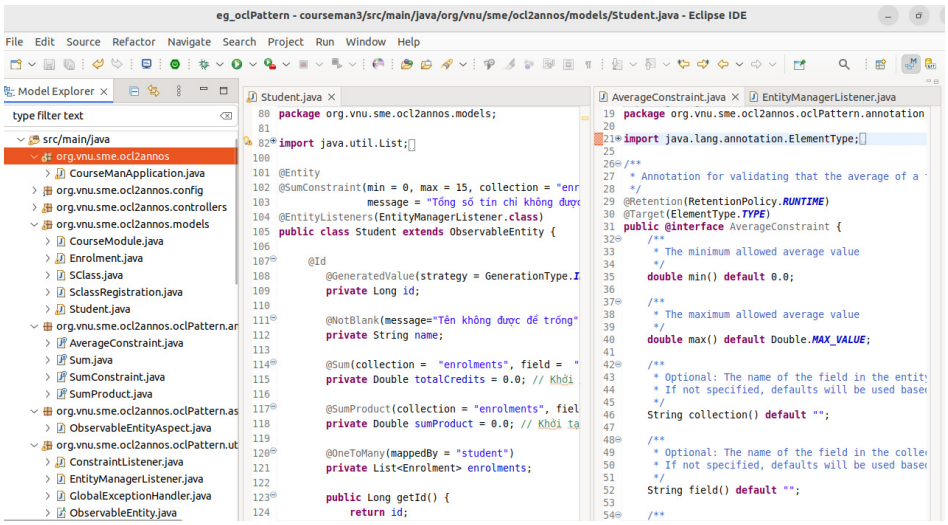

Fig. 7: Tool support realization and usability of a CAP framework.

unified domain model implemented in Java.

We provide an example of validating the total number of credits for each student's enrollment. Suppose the total registered credits must not exceed 15. At the top of Fig. 8, (1) shows the result where the student "Tran Van Anh" has registered for 14 credits, and (2) displays the error message triggered when this student attempts to register for more than the allowed credits. When performing the save operation to enroll in a new course, the system automatically activates the validation mechanism according to the method we proposed.

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 [40] 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 CAP for modeling domain concepts compared to existing DDD methods?

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

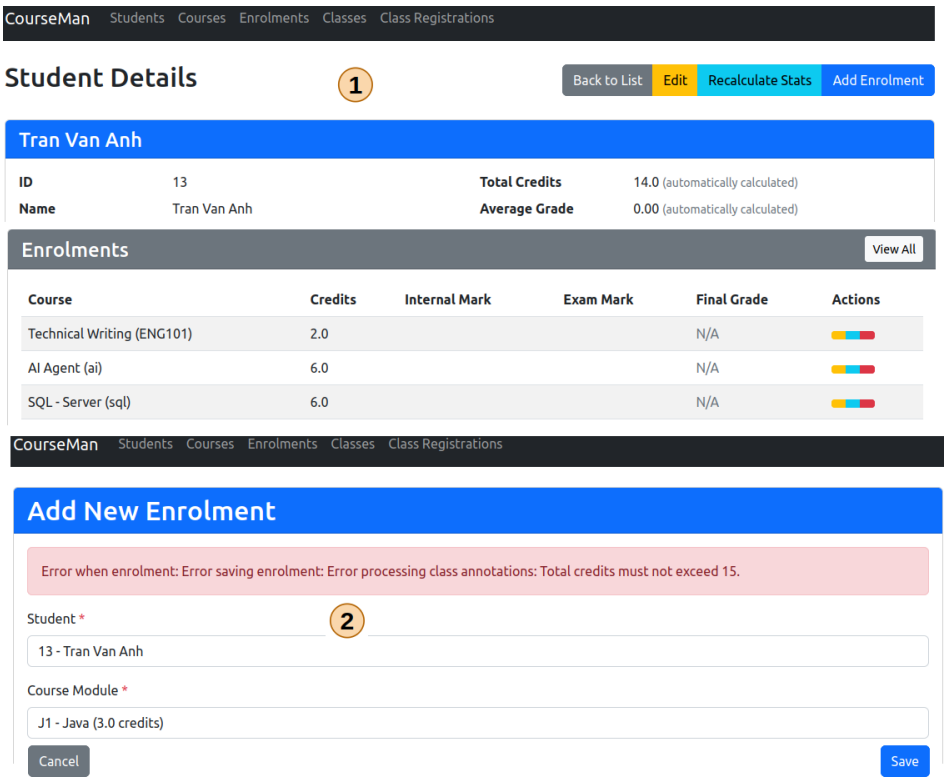

Fig. 8: The GUI of the CourseMan software generated by the tool.

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 CAP 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 the DCSL extension and its annotation-based enhancements with commonly used third-party annotation sets, including OpenXAVA [20], Apache Causeway [19], and Actifsource [41].

Expressiveness. We use CAP's terminology as the basis for evaluation and analyzed the relevant technical documentation of OpenXAVA, Apache Causeway and Actifsource to identify the language constructs. An additional analysis was conducted to quantify the correspondences.

We established 12 criteria based on the ability to use annotations to specify and enforce aspects of the domain model. We then evaluated our tool and the

third-party tools–OpenXAVA, Apache Causeway, and Actifsource–using the levels `Yes` (fully supported), `Partial` (partially or indirectly supported), and `No` (not supported) across these 12 criteria:

(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 2 presents details of the evaluation based on the above criteria.

Table 2: Comparing the expressiveness of CAP to OpenXAVA, Apache Causeway, and Actifsource.

Criteria	**CAP**	**OpenXava**	**Apache Causeway**	**Actifsource**
Support for annotations for quantity constraints	Yes: @SizeConstraint	Partial: @Size	Partial: @CollectionLayout	Yes: Define multiplicity 0..5 in the model
Support for annotations for logical constraints	Yes: @Prerequisite-Constraint	Partial: @AssertTrue with check method	Partial: @Property(edit= Edit.DISABLED)	Yes: Define logical constraints in the model
Support for annotations for derived values	Yes: @SumProuct	Partial: @Calculation with formula	Partial: @Property	Yes: Define derived attributes in the model
Support for annotations for invariants	Yes: @Eligibility-Constraint	Partial: @AssertTrue with invariant check method	Partial: @Programmatic with invariant logic	Yes: Define invariants in the model
Automatic generation of validation code from annotations	Yes: Generate code from @SumConstraint to check quantity	Partial: Generate code from @Size to validate	Partial: Generate code from @CollectionLayout	Yes: Generate code from the model to enforce constraints
Support for listening and triggering automatic validation	Yes: Annotation triggers listener when enrollments change	No: Does not support automatic listening	No: Does not support automatic listening	Partial: Depends on code generation tool
Integration capability with OOPL	Yes: Annotation in Java for Student class	Yes: Annotation in Java for entity	Yes: Annotation in Java for domain object	Yes: Annotation in generated Java code
Support for complex expressions (forAll, exists)	Yes: support complex expressions	No: Does not support complex expressions	No: Does not support complex expressions	Partial: Can be defined in the model
Support for time constraints	Yes: @TimeConstraint	Partial: @Future or @Past on date	Partial: @Property with time logic	Yes: Define time constraints in the model
Customization and extension capability for annotations	Yes: @Structural-Constraint defined by user	Partial: @Stereotype for extension	Partial: @Mixin for extension	Yes: User-defined annotations in the model
Integration with development tools (IDE)	Partial: Integration with IDE for auto-completion of annotations	Yes: Well integrated with Eclipse, NetBeans	Yes: Well integrated with IntelliJ, Eclipse	Yes: Tightly integrated with Eclipse
Support for multiple programming languages	Partial: Mainly for Java, can be extended	No: Only supports Java	No: Only supports Java	Partial: Supports Java and C++

Required coding level. A systematic approach is used based on five 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.

We use three rating levels for the above five criteria: Good, Average, and Low. Table 3 summarizes the comparison results of CAP with OpenXAVA, Apache Causeway, and Actifsource based on these five criteria. For details, see Appendix Appendix A.

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

Tool	**Required coding level**
CAP	Good: Reduces coding through automation and support for complex constraints via annotations. However, flexibility and customization are still issues.
OpenXava	Average: Reduces coding for basic cases but requires manual code for complex logic.
Apache Causeway	Average: Good automation but requires manual code for complex requirements.
Actifsource	Good: High automation, but the learning curve may impact initial effort.

Effectiveness. We use five 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: Average, Good, and Excellent. Table 4 summarizes the comparison results of CAP with OpenXAVA, Apache Causeway, and Actifsource based on these five criteria.

Table 4: Comparing the effectiveness of CAP to OpenXAVA, Apache Causeway, and Actifsource.

Criteria	**CAP**	**OpenXava**	**Apache Causeway**	**Actifsource**
Domain modeling capability	Good: Direct, rich annotations	Good: Basic annotations	Good: Basic annotations	Excellent: Detailed modeling
Support for complex constraints	Excellent: OCL via annotations	Average: Basic, requires additional code	Average: Basic, requires additional code	Good: Defined in the model
Consistency between model and code	Good: Annotations embedded in code	Good: Synchronized annotations, but limited	Good: Synchronized annotations, limited	Excellent: Code generated from the model
Effectiveness in automation	Excellent: Automatic validation code generation	Good: Generates interface, storage	Good: Generates interface, API	Excellent: Generates entire code
Scalability and maintainability	Average: Adding annotations requires code construction	Average: Difficult with complex logic	Average: Difficult with customization	Excellent: Easy model adjustments

6.3. *CAP applicability in real-world software engineering*

This section presents the results of our experiments on CourseMan and OrderMan, 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 and OrderMan. The results are summarized in Table 5.

In CourseMan, 73 constraints were analyzed and classified into 14 groups. CAP could formally represent 62 of these constraints, covering 11 groups. The remaining 11 constraints were outside the current scope of CAP, as they involved dynamic conditions dependent on runtime states, events, or contextual triggers. These include pre-/post-condition constraints, access control constraints, and automatic system reaction constraints. Overall, the CAP approach addressed 84.9% of the constraints in CourseMan.

For OrderMan, we analyzed 71 constraints spanning the full spectrum from basic structural integrity rules to complex business logic and system behaviors. CAP were applicable to 60 constraints, with the remaining 11 sharing the same limitations observed in CourseMan. The applicability rate in this case was 84.5%.

Table 5: Constraint groups and CAP patterns

Constraint Group	**CAP Pattern**	**Applicability of CAP**	
		CourseMan	OrderMan
Well-formedness constraints	DCSL 1	11/11	11/11
Quantitative limits	SumConstraint	6/6	3/3
Dependency and prerequisite constraints	PrerequisiteConstraint	5/5	3/3
Scheduling and conflict constraints	ScheduleConstraint	4/4	3/3
Entity qualification constraints	EligibilityConstraint	5/5	3/3
Retry or reattempt rules	RetakeConstraint	4/4	2/2
Capacity constraints	SizeConstraint	5/5	4/4
Time and deadline constraints	TimeConstraint	8/8	4/4
Computation and derivation rules	SumProduct	5/5	3/3
Status-based constraints	StatusConstraint	5/5	4/4
Organizational structure constraints	StructuralConstraint	4/4	8/8
Pre-/post-condition constraints	✘	0/5	0/5
Access control constraints	✘	0/4	0/3
Automated reaction constraints	✘	0/2	0/3
	Total	**62/73**	**60/71**

6.4. *Discussion*

This section discusses potential threats to the validity of both our proposed method and its evaluation. We will make remarks on the modularity and implementation of CAP, its role in software design, and considerations related to expressiveness, required coding effort, effectiveness, and applicability.

In the design and implementation of

Within software design, CAP extends DDD by embedding constraint definitions directly into the domain model, thereby consolidating business rules at a single authoritative source. This centralization reduces specification redundancy, mitigates inconsistencies, and lowers long-term maintenance costs. Furthermore, the integration of the observable pattern facilitates continuous, real-time validation, ensuring that any constraint violation is detected and addressed immediately, thus preserving the semantic integrity of the system throughout its lifecycle.

We discuss and point out the specific threats that affect our method as follows:

Expressiveness, While CAP effectively captures many OCL constraint patterns, it has limitations. Complex temporal logic, higher-order operations, and certain quantifiers may be challenging to represent using annotations. Additionally, discrepancies between OCL's mathematically precise semantics and Java's pragmatic type system may result in approximate constraint enforcement.

Required coding level, CAP significantly reduces the amount of manual validation code needed, but it introduces a new requirement for developers to understand OCL concepts and the pattern mapping system. Setting up the annotation processors and observer pattern infrastructure requires initial investment, though this is offset by reduced maintenance costs over time. Organizations must consider this learning curve when adopting the framework.

Effectiveness, The effectiveness of CAP depends on several factors. Performance may be impacted in large-scale systems due to reflection and runtime evaluation of constraints. While our implementation in CourseMan demonstrated good performance characteristics for typical educational scenarios, highly transaction-intensive systems might experience bottlenecks. Additionally, the framework's effectiveness relies on comprehensive test coverage to ensure all constraints are properly defined and enforced.

Applicability While our study successfully defined and implemented eleven CAP patterns encompassing major business rules and constraints, and demonstrated their effective application to both CourseMan and OrderMan, the broader applicability of the approach across diverse domains depends on extending the CAP library with additional domain-specific patterns. Such an extension would enable CAP to capture constraints and semantics unique to new application areas without requiring significant methodological changes. Furthermore, the generation of executable software artifacts from CAP-based models also relies on the development of appropriate runtime components—particularly listeners, processors, and validators—that operationalize the constraint logic during execution. Thus,

while the current results confirm the feasibility and robustness of CAP in representative DDD projects, their adoption in other domains will benefit from systematic pattern expansion and supporting component development.

7. 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 [42] 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 [43] 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 [44, 45] 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 [14–17] use OCL combined with annotations to represent domain models in a DDD approach. The study by [46] 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 [45]; (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.

8. Conclusion and Future Work

In this paper, we introduce a novel constraint annotation patterns method (CAP) based on annotations to capture complex OCL constraints specification, enhancing domain model representation and automating the software development process to address key features of DDD: feasibility and satisfiability. Through the development and application of the CAP framework and the JDA framework, complex constraints are processed via integrated annotations, supporting the automatic generation of validation code, reducing errors, and improving development efficiency. Designed for applicability across various technical domains, the approach incorporates an integrated listening mechanism to ensure the continuous correctness of the domain model. Furthermore, it serves as a bridge between declarative specifications and imperative implementations, enabling a fully executable model while minimizing inconsistencies and promoting reusability

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.

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*, 2014).
[3] V. Vernon, *Implementing domain-driven design*, 1 er edn. (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 (November 2023), arXiv:2310.01905 [cs].
[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* **11**(2) (2024) 79–83, Number: 2.
[6] M. Brambilla, J. Cabot and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd edn. (Morgan&Claypool, 2017).
[7] J. Griffin, Domain-Driven Laravel, Learn to Implement Domain-Driven Design Using Laravel (January 2021).
[8] OMG, *Unified Modeling Language 2.5.1* (Object Management Group, 2017).
[9] M. Owashi, K. Okano and S. Kusumoto, A Translation Method from OCL into JML by Translating the Iterate Feature into Java Methods, *Computer Software* **27**(2) (2010) Publisher: Japan Society for Software Science and Technology.
[10] E. V. Sunitha and P. Samuel, Object constraint language for code generation from activity models, *Information and Software Technology* **103** (November 2018) 92–111.
[11] M. Fowler, *Domain-Specific Languages*, 1st edn. (Addison-Wesley Professional, September 2010).
[12] Andrzej Wasowski and Thorsten Berger, *Domain-Specific Languages Effective Modeling, Automation, and Reuse* (Springer Cham, 2023).

[13] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser and G. Wachsmuth, *DSL Engineering - Designing, implementing and using domain-specific languages* (M Volter / DSLBook.org, Stuttgart, Germany, 2013).

[14] 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* **45** (April 2016) 164–190.

[15] 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.

[16] 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.

[17] 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* **39** (October 2013) 1376–1402, Conference Name: IEEE Transactions on Software Engineering.

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

[19] Dan Haywood, Apache Isis - Developing Domain-Driven Java Apps, *Methods & Tools: Practical knowledge source for software development professionals* **21**(2) (2013) 40–59.

[20] J. Paniza, *Learn OpenXava by example*, 1.1 edn. (CreateSpace, 2011).

[21] F. U. Haq and J. Cabot, B-OCL: An Object Constraint Language Interpreter in Python (March 2025), arXiv:2503.00944 [cs].

[22] D. M. Le, D.-H. Dang and V.-H. Nguyen, On domain driven design using annotation-based domain specific language, *Computer Languages, Systems & Structures* **54** (December 2018) 199–235.

[23] D.-H. Dang, D. M. Le and V.-V. Le, AGL: Incorporating behavioral aspects into domain-driven design, *Information and Software Technology* **163** (November 2023) p. 107284.

[24] 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, November 2023), pp. 25–34.

[25] 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)*, October 2023, pp. 1–6. ISSN: 2694-4804.

[26] A. Wasowski and T. Berger, Internal Domain-Specific Languages, in *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*, (Springer, Cham, 2023) pp. 357–394.

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

[28] J. A. Hoffer, J. George and J. A. Valacich, *Modern Systems Analysis and Design*, 7a edn. (Pearson, 2013).

[29] D.-H. Dang and J. Cabot, On Automating Inference of OCL Constraints from Counterexamples and Examples, in *Knowledge and Systems Engineering*, eds. V.-H. Nguyen, A.-C. Le and V.-N. Huynh (Springer International Publishing, Cham, 2015), pp. 219–231.

[30] S. Chippagiri, A Comprehensive Review of Software Design Patterns: Applications and Future Direction (2025).

[31] R. Mzid, S. Selvi and M. Abid, Research Landscape of Patterns in Software Engineering: Taxonomy, State-of-the-Art, and Future Directions, *SN Computer*

Science **5** (April 2024) 1–24, Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer Nature Singapore.

[32] J. Willans, P. Sammut, G. Maskeri, A. Evans and A. Clark, Defining OCL expressions using templates. (2002), Publisher: King's College London.

[33] A. Hamie, Constraint specifications using patterns in OCL, *International Journal on Computer Science and Information Systems* **8** (January 2013).

[34] J. Cabot, R. Clarisó and D. Riera, On the verification of UML/OCL class diagrams using constraint programming, *Journal of Systems and Software* **93** (July 2014) 1–23.

[35] A. Hamie, Pattern-based mapping of OCL specifications to JML contracts, in *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, January 2014, pp. 193–200.

[36] M. Voelter, Language and IDE Modularization and Composition with MPS *Lecture Notes in Computer Science (LNPSE)* **7680** (Springer, Berlin, Heidelberg, 2013) pp. 383–430.

[37] 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)*, December 2024, pp. 424–428. ISSN: 2473-0130.

[38] B. Demuth, H. Hussmann and A. Konermann, Generation of an OCL 2.0 Parser, in *Proceedings of the MoDELS '05 Conference Workshop on Tool Support for OCL and Related Formalisms-Needs and Trends. EPFL, Technical Report LGL-REPORT-2005-001*, 2005, pp. 38–52.

[39] 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)*, December 2019, pp. 399–406.

[40] Thakur and U. Pandey, The Role of Model-View Controller in Object Oriented Software Development, *Nepal Journal of Multidisciplinary Research* **2** (November 2019) 1–6.

[41] *Actifsource* (Actifsource AG, Switzerland - all rights reserved., 2017).

[42] J. Cabot and E. Teniente, Transformation techniques for OCL constraints, *Science of Computer Programming* **68** (October 2007) 179–195.

[43] 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.

[44] L. Hamann, O. Hofrichter and M. Gogolla, OCL-Based Runtime Monitoring of Applications with Protocol State Machines, in *Modelling Foundations and Applications*, **7349** (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012) pp. 384–399. Series Title: Lecture Notes in Computer Science.

[45] J. Chimiak-Opoka, B. Demuth, A. Awenius, D. Chiorean, S. Gabel, L. Hamann and E. Willink, OCL tools report based on the ide4OCL feature model, *Electronic Communications of the EASST* **44** (2011).

[46] 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*, (ACM, Saint-malo France, October 2016), pp. 65–75.

Appendix A. Required coding level:

Table 6: Comparing the required coding level of CAP to OpenXAVA, Apache Causeway, and Actifsource.

Criteria	**CAP**	**OpenXava**	**Apache Causeway**	**Actifsource**
Automation of Code Generation	Good: Annotations automatically generate validation code for constraints.	Good: Generates interface and storage code from annotated Java classes.	Good: Generates interface and storage from annotated domain objects.	Good: Model-based approach generates most code from the domain model.
Support for Complex Constraints	Good: Supports complex OCL-like expressions via annotations, no manual code needed.	Average: Supports basic constraints (e.g., @Size, @Min); complex constraints require custom methods.	Average: Basic constraints via annotations; complex logic requires custom methods or mixins.	Good: Complex constraints are defined in the model and executed in generated code.
Integration with Domain Model	Average: Annotations are embedded directly into code, tightly integrated with the domain model.	Good: Annotations applied to entity classes, well integrated with the model.	Good: Annotations applied to domain objects, ensuring solid integration.	Good: Model-centric approach ensures complete integration through generated code.
Learning Curve	Average: Requires learning OCL syntax and annotation usage.	Low: Familiar to Java developers, especially those with JPA experience.	Average: Requires understanding the tool's specific programming model.	Good: Requires knowledge of model-driven development, quite challenging to learn.

Criteria	**ModelPattern**	**OpenXava**	**Apache Causeway**	**Actifsource**
Flexibility and Customization	Good: Allows manual coding in special cases, but automates most tasks.	Average: Customizable but often requires additional manual code.	Average: Customization through mixins and layouts, but complex cases need extra code.	Average: Customization may require model adjustments or template code modifications.