A Method for Composing Concerns into a Unified Domain Model in Domain-Driven Design

Van-Vinh Le[1,2], Nhat-Hoang Nguyen[2], Duc-Quyen Nguyen[2], and Duc-Hanh Dang[2]⋆

[1] Vinh University of Technology Education `levanvinh@vuted.edu.vn`
[2] Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Viet Nam
`{24025170, 21020388, hanhdd}@vnu.edu.vn`

Abstract. Domain-Driven Design (DDD) emphasizes iterative development around a rich domain model to align developers and domain experts. While ubiquitous language and Domain-Specific Languages (DSLs) improve expressiveness and maintainability, modern systems often require multiple heterogeneous DSLs to cover diverse concerns. Existing DDD approaches, however, lack systematic methods to compose such DSLs, resulting in fragmented models and limited automation. Although meta-modeling offers a standard way to define DSLs, it is often rigid and framework-dependent. This paper introduces a novel method for composing heterogeneous concern DSLs into a unified domain model within DDD. Each DSL is defined with consistent syntax and formal semantics, and integrated via an annotation-based composition mechanism at the abstract syntax tree (AST) level. This ensures concern orthogonality, model cohesion, and supports consistency checking, automated code generation, and traceability. The approach is implemented using JetBrains MPS and the JDA framework, and validated through representative case studies, advancing modular and executable domain modeling for complex systems.

Keywords: Domain-Driven Design · Domain-Specific Language · Domain Model · MPS · Metamodeling

1 Introduction

The increasing complexity, scale, and heterogeneity of modern software systems have intensified the need for methodologies that effectively bridge domain knowledge and technical implementation. Domain-Driven Design (DDD) has emerged as a prominent approach to this challenge, advocating iterative development around a semantically rich domain model that encapsulates the core logic and rules of the problem domain [10,17]. Central to DDD is the consistent use of a Ubiquitous Language (UL), which fosters effective communication and

⋆ Corresponding author

shared understanding between domain experts and developers throughout the software lifecycle. To operationalize the UL and align it with implementation artifacts, Domain-Specific Languages (DSLs) have been widely adopted as expressive means of encoding domain concepts directly in source code [12]. DSLs tightly couple design models with implementation, thereby improving software correctness, maintainability, and extensibility [4, 11, 25, 29]. Defining and integrating multiple concern DSLs—such as those for business logic, security, user interfaces, and performance—is essential for capturing diverse requirements in complex software systems [13, 16].

Current approaches often address concerns in isolation, leading to tight coupling and limited extensibility. Recent efforts [22] have focused on defining metamodels based on UML metaconcepts to support the composition and unification of concern-specific DSLs. Within the method, the abstract syntax is defined through meta-concepts in an external DSL, while a one-to-one mapping to an annotation-based DSL provides an internal DSL representation within the host language. Nevertheless, significant limitations remain: integration processes are typically semi-automated and error-prone, rely heavily on high-quality mappings between DSLs, and offer limited support for complex or cross-cutting concerns. Scalability also becomes problematic as the number and complexity of concerns increase, while existing approaches still fail to capture sophisticated structural and behavioral aspects [7, 21].

In this paper, we propose a methodology for composing heterogeneous concern DSLs into a unified domain model. The basic idea is to adopt an Abstract Syntax Tree (AST)-based method, which emphasizes simplicity, direct manipulation without parsing, support for multiple syntactic views, and provides a solid basis for consistency checking, executable code generation, and end-to-end traceability. Each DSL is uniformly defined in terms of abstract syntax (AS), concrete syntax (CS), and semantics, establishing a coherent foundation for cross-concern integration. Unification is achieved through an annotation-based composition mechanism at the AST level, which interrelates DSLs and binds their semantics to core domain elements. This ensures the orthogonality of concern DSLs and preserves the cohesion of the unified domain model. We implement and validate the approach using the projectional editing capabilities of JetBrains MPS [3] and the DDD framework JDA [9], demonstrating its feasibility and effectiveness through real-world case studies.

In brief, the contributions of this paper are as follows:

– A novel AST-based methodology for composing heterogeneous concern DSLs into the unified domain model, referred to as UDML.
– A complete definition of the syntax and semantics of UDML, establishing a coherent modeling foundation for cross-concern integration.
– A tool-supported implementation on the JetBrains MPS platform, advancing the state of the art in modular and extensible domain modeling.

The rest of the paper is organized as follows: Section 2 motivates our work with examples and background. Section 3 explains the basic idea of our approach. Section 4 provides the syntax and semantics for UDML and the mechanism to compose concerns. Tool support and experiments are explained in Section 5, which discusses the proposed language. Section 6 surveys related work. Finally, Section 7 concludes the paper with a discussion of future work.

2 Motivating Example and Background

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

2.1 Motivating Example

We use the `CourseMan` domain, adapted from [21], as a motivating example. Fig. 1 presents the `CourseMan` domain model, based on OMG/UML, combining a class diagram with concern DSLs. It includes three main classes `Student`, `CourseModule`, `SClass` and two association classes `SClassRegistration` and `Enrolment`. The dashed shapes labeled (1) represent the DCSL concern, which captures the structural aspect through four concepts `DClass`, `DAttr`, `DOpt`, `DAssoc` defining entities and associations (e.g., **`enrolsIn`**, **`registersTo`**). The dashed shapes labeled (2) denote the AGL concern, which specifies behavioral aspects through `AGraph` defining the reachable state, while the star-like shape '**?**' indicates extensibility for future concerns. The dashed shapes labeled (2) denote the AGL concern, which specifies behavioral aspects through `AGraph` defining reachable states, while the star-like shape '**?**' indicates extensibility for future concerns. These concerns must be systematically composed into the CourseMan problem domain to facilitate the integration and evolution of concern DSLs in a modular and scalable manner.

The `CourseMan` system exemplifies a domain characterized by intertwined structural and behavioral concerns. Although concern-specif DSLs can effectively capture their respective aspects, they often lack modularity, expressiveness, and extensibility, which hinders systematic integration and evolution. For example, DCSL [21] supports structural modeling but provides little support for behaviors, roles, constraints, or security, and the addition of further DSLs only amplifies overlaps and inconsistencies. This underscores the need for a unified modeling approach that enables separation of concerns, consistent integration, and continuous evolution of heterogeneous DSLs within a unified executable model. In this context, Fig. 1 illustrates the **`refCons`** relation, which establishes a cross-reference between the concern DSLs DCSL and AGL.

2.2 Background

To leverage domain expertise effectively, we revisit foundational concepts that underpin our concern-driven DSL integration approach.

Executable Domain Models in DDD. Domain-Driven Design (DDD) [10] advocates iterative development around a semantically rich domain model that encapsulates core logic and business rules. This model establishes a shared foundation between domain experts and developers through a consistent Ubiquitous Language (UL) [17, 32]. Domain-Specific Languages (DSLs) reinforce this alignment by encoding domain knowledge directly into software artifacts [2, 12], enhancing correctness, maintainability, and extensibility.

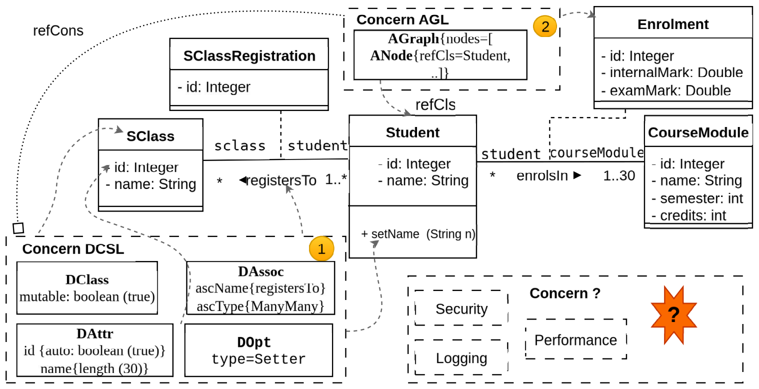


Fig. 1. UML class diagram and concern DSLs in CourseMan.

Existing DDD approaches support domain modeling but often neglect the systematic formation and management of software modules derived from domain models [6, 15, 16, 26], leading to the absence of robust methodologies for their development and evolution. A key advancement in DDD is the executable domain model, which unifies descriptive and operational semantics [34]. By integrating concerns such as business logic and security into a runnable specification, executable models enable automation, verification, and adaptability [1]. However, achieving this vision requires automated composition of heterogeneous concerns—an ongoing challenge in DDD research.

Composing Concern DSLs in UDML. Composing concern DSLs can be approached through two main methods: the AST-based method and the meta-modeling method. The AST-based method represents models as hierarchical trees of language concepts, where nodes denote concern elements and edges capture composition or reference relations [35]. It enables direct manipulation of model instances without parsing, supports multiple syntactic views on a shared structure, and provides an operational basis for consistency checking, code generation, and traceability.

The meta-modeling method, in contrast, defines DSLs at a higher abstraction level using metaclasses, attributes, and relations, offering standardized mechanisms for validation, transformation, and tool interoperability [19]. Our work [22] applies this method by extending UML metaconcepts to integrate concern DSLs into a cohesive metamodel. Each concern is modeled with a dedicated DSL, such as DCSL for structural aspects [21], AGL for behavioral logic [7], while additional DSLs can be incorporated as needed. However, meta-modeling approaches face significant limitations: integration processes often rely on complex and fragile mappings, tool support is highly framework-dependent, and extensibility is hindered when the number or heterogeneity of concerns increases. These drawbacks reduce scalability and complicate the handling of cross-cutting concerns in practice.

Therefore, this work aims to adopt the AST-based method, leveraging its strengths in flexibility, direct manipulation, and multi-view integration to compose heterogeneous concern DSLs into the unified domain model within the context of DDD. The goal is to improve extensibility and expressiveness in complex software systems.

3 Overview of Our Approach

This section introduces our approach to composing concern DSLs into a unified domain model. Figure 2 illustrates a four-step iterative process for integrating and refining these DSLs toward automatic software prototype generation.

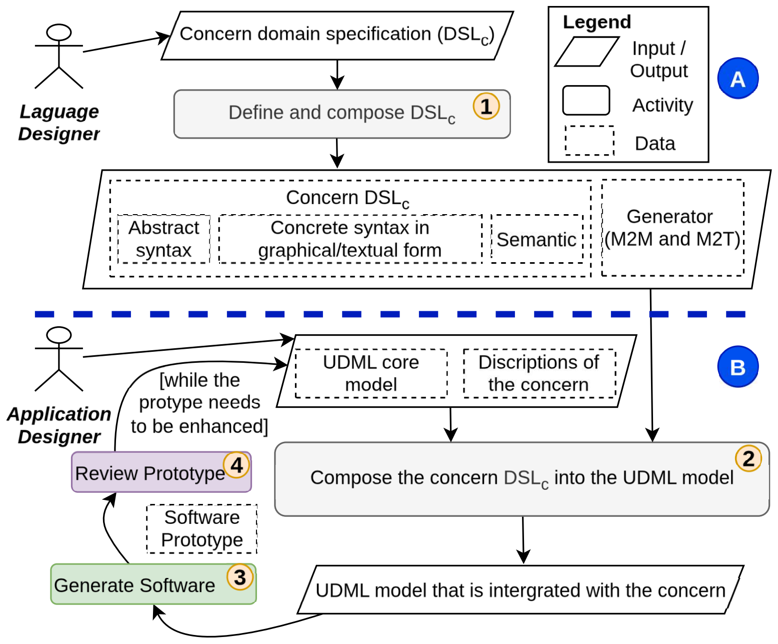

Laguage Designer
Concern domain specification (DSLc)
Define and compose DSLc 1
Legend
Input / Output
Activity
Data
A
Concern DSLc
Abstract syntax
Concrete syntax in graphical/textual form
Semantic
Generator (M2M and M2T)
Application Designer
UDML core model
Discriptions of the concern
B
[while the protype needs to be enhanced]
Review Prototype 4
Compose the concern DSLc into the UDML model 2
Software Prototype
Generate Software 3
UDML model that is intergrated with the concern

Fig. 2. An overview of our approach, organized into two main layers: (A) the language design level and (B) the language application level.

First, at the language design level (label A), the designer specifies the concern domain using a domain-specific language (DSL_c) as input. For each DSL_c, the AS, CS, and semantics are formally defined, resulting in a DSL_c that precisely captures the target concern. *Second*, at the application level (label B), the designer uses the outputs from step one and existing UDML core model. Along with descriptions of new or changing concern requirements, these inputs help compose the concern DSLs (DSL_c) into the UDML model for coherent integration. Textual or graphical CS may be used to construct a detailed model that represents the specific concern instance. The output of this step is a UDML model fully integrated with the given concern. *Third*, the integrated UDML model serves as the foundation for generating production software. The resulting software artifact is then evaluated by the designer to gather feedback. *Finally*, if feedback is provided, both the UDML model and the corresponding concern specification are updated accordingly. The process then repeats, supporting a continuous improvement cycle until the software system meets the specified requirements.

4 Composing Concern DSLs

This section defines the syntax, semantics of UDML and introduces the algorithm for composing heterogeneous DSLs into a unified domain model.

4.1 Defining the Syntax and Semantics of UDML

Our approach to defining the UDML adopts a structured three-layer framework comprising abstract syntax, concrete syntax, and formal semantics, complemented by an integration strategy.

Definition 1. *The **UDML** language is constructed from a core UDML and a set of concern DSLs* $\{DSL_i\}_{i=1}^{n}$*, where each* DSL_i *describes a distinct aspect of the system. Each DSL is specified by a triplet* (AS_i, CS_i, Sem_i)*, where* AS_i*,* CS_i*, and* Sem_i *represent the abstract syntax, the concrete syntax, and the semantics of the **UDML**, respectively.*

Abstract syntax definition (AS): The AS of UDML is formalized as a hierarchical tree structure where each node denotes an instance of a language concept, edges represent composition or reference relations, and attributes capture local properties of nodes. Concern DSLs extend the tree incrementally by introducing new concepts and relations while preserving the common backbone defined in the UDML core.

Definition 2. *Let* C_{UDML} *be the set of meta-concepts of the UDML and* A_{UDML} *the set of attributes applicable to these concepts. For each concept* $c \in C_{UDML}$*, let* $A(c) \subseteq A_{UDML}$ *denote the set of attributes defined for c. The **AS** of UDML is defined as the tuple* $AS_{UDML} = (N, root, child, refCons, label, attr, P_{UDML})$*, where:*

- N*: A finite set of nodes.*
- $root \in N$*: The root node representing the unified domain model, instantiated in UDML as* `DomainModel`*.*
- $child \subseteq N \times N$*: A tree relation specifying the ordered children of each node.*
- $refCons \subseteq N \times N$*: A set of cross-reference edges linking nodes across concerns.*
- $label : N \rightarrow C_{UDML}$*: A labeling function assigning each node to a meta-concept.*
- $attr : N \rightarrow (A(label(n)) \rightarrow V)$*: An attribute function assigning values to the attributes of the node's concept.*
- P_{UDML}*: A set of well-formedness rules and constraints on the tree structure.*

Example. Fig. 3 shows parts of the AS of DCSL, AGL, and UDML.
DSL for the structural aspect concern - DCSL

```
   root = DomainModelDCSL,
child(DomainModelDCSL) = {DClass, DAssoc, DAttr, DOpt}
```

C_{DCSL} = {`DomainModelDCSL`, `DClass`, `DAssoc`, `DAttr`, `DOpt`}
A_{DCSL}= {`DClass(mutable)`, `DAttr(auto,length)`}
DSL for the behavioral aspect concern - AGL
`root = DomainModelAGL, child(DomainModelAGL) = {AGraph}, child(AGraph) = {ANode, AEdge, MAct}`
C_{AGL} = {`DomainModelAGL`, `AGraph`, `ANode`, `AEdge`, `MAct`}
A_{AGL} ={`AGraph(nodes)`, `ANode(label, refCls, outNodes, actSeq)`}
UDML AS for Composing Concern DSLs: DCSL and AGL:
`root = DomainModel, child(DomainModel) = {DClass, DAssoc, DAttr, DOpt, AGraph}, refCons(AGraph` $\rightarrow$ `DClass)`
$C_{UDML} = C_{DCSL} \cup C_{AGL} \cup$ {`Annotable`, `Concern`, `Annotation`, `UmlClass`}
$A_{UDML} = A_{DCSL} \cup A_{AGL}$.

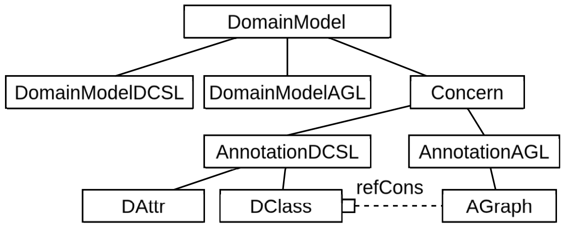


Fig. 3. UDML AS for composing concern DSLs: DCSL and AGL.

Concrete syntax definition (CS): The CS is defined as a projection of the AST into user-facing notations. Each syntactic view provides a domain-appropriate visualization while all views operate on the same underlying AST. This ensures consistency, usability for diverse stakeholders, and synchronization across notations.

Definition 3. *The **CS** of UDML is defined as the tuple* $CS_{UDML} = (V, M, \pi, H, P_{CS})$ *where:*

- *V: A set of visual symbols (graphical or textual primitives such as boxes, arrows, tables, tags).*
- *M: A set of mapping rules that associate AST nodes or edges to visual symbols.*
- *$\pi : N \cup (N \times N) \rightarrow V$: A projection function mapping AST nodes $n \in N$ or edges (n_i, n_j) to concrete symbols in V.*
- *H: A set of handlers supporting interactive editing actions (e.g., creation, modification, drag-and-drop, annotation).*
- *P_{CS}: A set of presentation constraints ensuring the validity and consistency of visualizations.*

In addition, the projection function can be indexed by view, i.e., $\pi_v : N \cup (N \times N) \rightarrow V$, where $v \in Views$, allowing multiple concrete notations for the same AST.

Formal semantics specification (Sem):
The semantics of UDML is defined by assigning meaning directly to nodes and

edges of the abstract syntax tree. Each concern DSL contributes a modular semantic mapping, and these mappings are composed along the AST structure.

Definition 4. *The **Sem** of UDML is defined by a function* $\langle Sem \rangle_{UDML} : N \cup (N \times N) \to D$ *where:*

- N*: the set of AST nodes.*
- $N \times N$*: the set of edges (child or reference relations).*
- D*: the semantic domain, defined as a product of concern domains:* $D = D_{dcsl} \times D_{agl} \times D_{sec} \times D_{perf} \times D_{log}$
- $\langle Sem \rangle_{UDML}$*: the semantics of a node* $n \in N$*, determined by the rules of its corresponding concern DSL.*
- $\langle (n_i, n_j) \rangle_{UDML}$*: the semantics of an edge, defined by how parent-child or cross-reference relations propagate semantics.*

The global semantics $\langle Sem \rangle_{UDML}$ is obtained by composing the semantics of all nodes and edges in the AST, ensuring consistency across concerns and enabling verification, simulation, and code generation. For instance, a checking rule validates that an `Annotation` applies only to a `Class`, reporting an error if violated.

```
if (!(node.target.isInstanceOf(concept<Class>)))
{    error("Target must be a Class."); }
```

The unification of concern DSLs into UDML is achieved through incremental composition with the tree-merging Algorithm 1. Concepts are integrated via inheritance, while annotations attach cross-cutting concerns. The full specification is in the technical report (GitHub[3]).

4.2 A Mechanism for Composing Concern DSLs in UDML

A unified AST is constructed from multiple concern DSLs, organized around the UDML Core and its extensions (e.g., DCSL, AGL). Each node denotes an instance of a metaconcept (e.g., `Student:DClass`), and edges capture composition or reference relations (e.g., `AGraph` $\to$ `DClass`). Algorithm 1 presents a tree-merging process that incrementally unifies concern DSLs into the UDML framework. The algorithm begins by initializing the UDML with a Core module, which serves as the foundation for all DSLs to inherit and extend (line 1). It then iterates over each DSL in the input set, merging them sequentially into the UDML core (line 2). The merger starts with the AS: for each concept in the current DSL, the concept is added to the UDML's AS if not already present (lines 3–15). Next, the CS is merged (lines 16–19), followed by the integration of formal semantics in a modular fashion, such as mapping AGL semantics to a state machine (lines 20-22). After all DSLs have been processed, the algorithm returns the unified UDML (line 24). This results in a consolidated, executable model that integrates diverse concerns while maintaining modularity and traceability.

[3] https://github.com/vinhskv/udml-syntax-soict2025/tree/main/technical/techRep.pdf

Algorithm 1 Tree-merging algorithm for integrating concern DSLs into UDML

```
Input: $D = \{DSL_1, DSL_2, \dots, DSL_n\}$: A set of concern DSLs, each specified as a
    triplet $(AS_i, CS_i, Sem_i)$
Output: $UDML_{unified}$: The unified UDML comprising a global $AS$, $CS$, and $Sem$
 1: Initialize: $AS_{UDML} \leftarrow AS_{core}$, $CS_{UDML} \leftarrow CS_{core}$, $Sem_{UDML} \leftarrow Sem_{core}$
 2: for each $DSL_i = (AS_i, CS_i, Sem_i) \in D$ do            ▷ //Merge Abstract Syntax
 3:     for each concept $c \in AS_i$ where $c \notin AS_{UDML}$ do
 4:         if $c$ extends core concept then
 5:             Integrate inheritance into $AS_{UDML}$
 6:         end if
 7:         Add attributes, relations, and constraints of $c$ to $AS_{UDML}$
 8:     end for                                          ▷ //Merge Edges (child, ref)
 9:     for each edge $(n_i, n_j) \in child_i \cup refCons_i$ do
10:         if $(n_i, n_j) \notin (child_{UDML} \cup refCons_{UDML})$ then
11:             Add edge $(n_i, n_j)$ into $AS_{UDML}$
12:         else
13:             Resolve conflicts by applying mapping or priority rules
14:         end if
15:     end for                                          ▷ //Merge Concrete Syntax
16:     for each new or extended concept $c$ do
17:         Add graphical or textual symbols for $c$ into $CS_{UDML}$
18:         Extend UI handlers (e.g., drag-drop, annotations, editing operations)
19:     end for                                          ▷ //Merge Semantics
20:     Integrate $Sem_i$ into $Sem_{UDML}$ modularly
21:     Compose new constraints into $P_{UDML}$
22:     Check cross-concern consistency between $Sem_i$ and $Sem_{UDML}$
23: end for
24: return $UDML_{unified} = (AS_{UDML}, CS_{UDML}, Sem_{UDML})$
```

5 Tool Support and Experiments

This session presents our methodology for unifying multiple concern-specific DSLs into a single AST of UDML, along with tool support and experiments.

5.1 Unifying Multiple Concern-Specific DSLs into a Single AST of UDML

The proposed methodology establishes a unified modeling approach that integrates the multiple concern-specific DSLs described in Section 4 into a single AST. The methodology relies on projectional editing and language composition mechanisms to achieve seamless interoperability across DSLs that address different concerns.

The significance of this methodology lies in its ability to reduce redundancy, ensure consistency across structural and non-structural aspects of domain models, and provide a coherent foundation for model-driven development. By unifying structural, behavioral, security, performance, and logging concerns into a single AST, the approach enables holistic reasoning, validation, and automated code generation across all domain aspects.

a) Core Structure of the AST: UDML Core
Metaconcepts. The UDML core defines the foundational concepts shared by all concern-specific DSLs:

- **DomainModel**: the root of the AST.
- **Annotable**: model elements that can be annotated.
- **Concern**: specific cross-cutting perspectives.
- **Annotation**: concern-specific semantics attached to annotable elements.
- **UmlModel**: inherits from UML class diagrams (Class, Association, Attribute, Operation).

Principle. Each concern-specific DSL must extend or compose its metaconcepts from the UDML core. This ensures that all DSLs remain interoperable and anchored within a common AST structure.

b) Concern-Specific DSL Integration: Each concern-specific DSL is integrated into UDML by extending the UDML core and introducing additional metaconcepts.
Structural DSL (DCSL)

- *Metaconcepts*: DomainModelDcsl, ConcernDcsl, AnnotationDcsl, DClass, DAttr, DAssoc, DOpt.
- *Integration*: DClass, DAttr, and DAssoc are linked to UML Class, Attribute, and Association, respectively. DOpt connects to UML Operation.
- *Mechanism*: Structural definitions are expressed as annotations on UML elements, thereby enabling reusability and alignment with object-oriented concepts.

Behavioral DSL (AGL)

- *Metaconcepts*: DomainModelAgl, ConcernAgl, AnnotationAgl, ActivityDomainModel, ActivityGraph, Node, Edge, ModuleAct, PreState, PostState.
- *Integration*: Activity graphs reference structural entities from DCSL to bind behavior to classes and operations.
- *Mechanism*: Behaviors are modeled as nodes and edges in activity graphs, with annotations linking them back to domain classes, ensuring traceability between structure and behavior.

Security DSL (RBAC)

- *Metaconcepts*: DomainModelRbac, ConcernRbac, AnnotationRbac, User, Role, Permission, Session, SeparationOfDuty.
- *Integration*: Roles and permissions are attached as annotations on annotable elements such as classes and operations.

– *Mechanism*: RBAC semantics are expressed through roles referencing users and permissions, with separation-of-duty constraints integrated via UDML annotations.

Performance DSL (PERF)

– *Metaconcepts*: DomainModelPerf, ConcernPerf, AnnotationPerf, PerfSpec, ServiceDemand, SLO, PerfAction.
– *Integration*: Performance specifications are linked to operations and services defined in DCSL and AGL.
– *Mechanism*: Service-level objectives (SLOs) and performance actions (insert, update, select, delete) are captured as annotations, allowing quantitative performance requirements to be validated against structural and behavioral models.

Logging DSL (LOG)

– *Metaconcepts*: DomainModelLog, ConcernLog, AnnotationLog, LogSpec, LogBefore, LogAfter, LogAction.
– *Integration*: Logging annotations are attached to operations or activity nodes.
– *Mechanism*: LogBefore and LogAfter specify hooks around actions, enabling transparent instrumentation of runtime execution without altering business logic.

Steps for Unification: The methodology follows a systematic six-step process:

1. **Domain Model Initialization**: The UDML core is instantiated, and the structural backbone of the domain is defined using UML class diagrams.
2. **Concern DSL Binding**: Each concern-specific DSL is bound to UDML by extending the DomainModel, Concern, and Annotation metaconcepts.
3. **Annotation Attachment**: Annotations from each DSL are attached to annotable elements in the AST, ensuring that all concerns are anchored to the core model.
4. **AST Composition**: The unified AST is composed by integrating concern-specific nodes into a shared projectional model. Cross-references maintain semantic links across structural, behavioral, and non-functional concerns.
5. **Validation and Constraints**: Consistency checks are applied using OCL-like constraints and type system rules to ensure correctness across concerns. This includes verifying structural validity, behavioral soundness, and constraint compatibility (e.g., security policies vs. performance requirements).
6. **Code Generation and Execution**: Code generators translate the unified AST into executable artifacts: structural entities (Java classes), behavioral workflows (state machines), security configurations (RBAC policies), performance monitors (SLO checks), and logging aspects (AOP code).

5.2 Tool Support and Discussion

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 4:

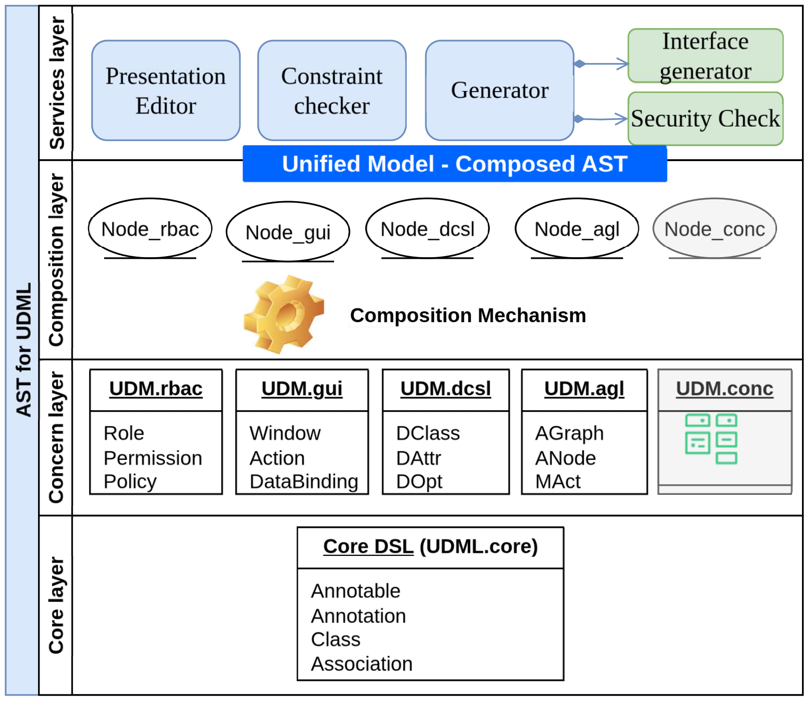


Fig. 4. Tool architecture of UDML with the proposed method.

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—`UDM.rbac`, `UDM.gui`, `UDM.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 [3, 33], 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).

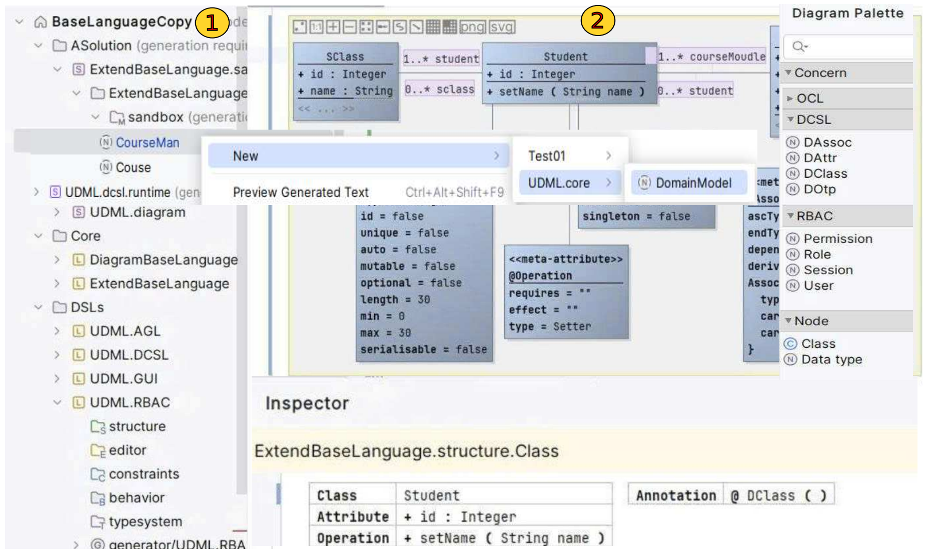
The tool facilitates the design, extension, and integration of concern DSLs, validating the approach's expressiveness, feasibility, and satisfiability. The practical applicability of our method is illustrated in Figure 5, and the implementation is publicly available at the Git repository[4].

Figure 5 illustrates our implementation. On the left, three core components are defined: (i) Instructor, specifying metaconcepts for UDML and concern DSLs; (ii) Editor, providing a graphical concrete syntax with drag-and-drop capabilities; and (iii) Generator, transforming the AST into executable code via the JDA framework [9]. We designed the UDML language as modular packages: `UDML.core` provides the integration foundation, while `UDML.dcsl`, `UDML.agl`, and `UDML.rbac` capture structural, behavioral, and security concerns, 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.

We applied the support tool to the CourseMan case study. Figure 5, label (2), illustrates the UML class diagram together with the integrated concern DSLs: DCSL, AGL, and RBAC. The tool then generates the corresponding source code. Finally, a software prototype is produced using the JDA framework, with further technical details provided in Figure 6.

In Figure 6, 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.

[4] https://github.com/vinhskv/udml-syntax-soict2025.git

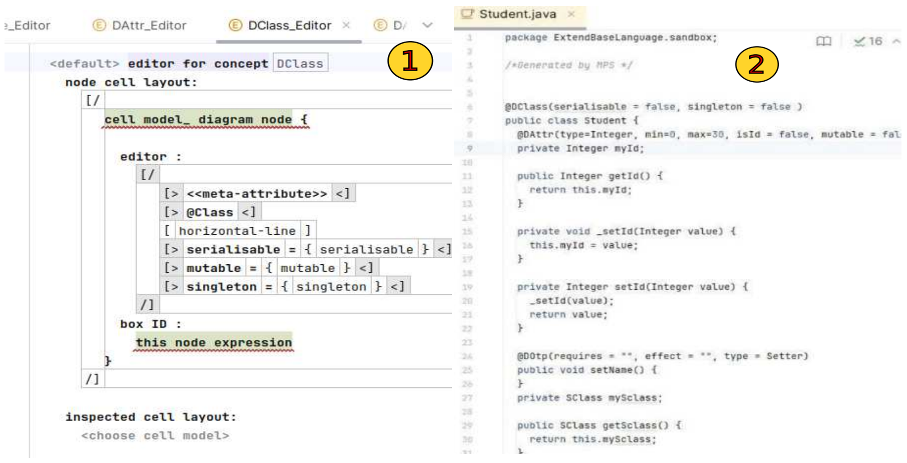


ExtendBaseLanguage.structure.Class

Class	Student
Attribute	+ id : Integer
Operation	+ setName (String name)

Annotation	@ DClass ( )

Fig. 5. MPS-based realization and practical integration of concern DSLs in UDML

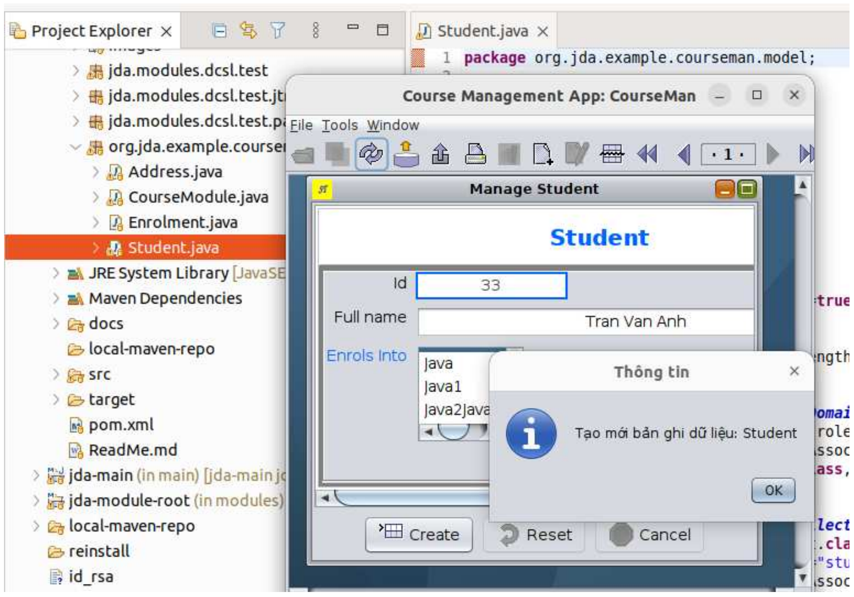


```
<default> editor for concept DClass
  node cell layout:
    [/
      cell model_ diagram node {

        editor :
          [/
            [> <<meta-attribute>> <]
            [> @Class <]
            [ horizontal-line ]
            [> serialisable = { serialisable } <]
            [> mutable = { mutable } <]
            [> singleton = { singleton } <]
          /]
        box ID :
          this node expression
      }
    /]

  inspected cell layout:
    <choose cell model>
```

```
package ExtendBaseLanguage.sandbox;

/*Generated by MPS */


@DClass(serialisable = false, singleton = false )
public class Student {
  @DAttr(type=Integer, min=0, max=30, isId = false, mutable = fal
  private Integer myId;

  public Integer getId() {
    return this.myId;
  }

  private void _setId(Integer value) {
    this.myId = value;
  }

  private Integer setId(Integer value) {
    _setId(value);
    return value;
  }

  @DOtp(requires = "", effect = "", type = Setter)
  public void setName() {
  }
  private SClass mySclass;

  public SClass getSclass() {
    return this.mySclass;
  }
```

Fig. 6. MPS-based Realization and Code Generation via the JDA Framework

Figure 7 illustrates the interface of the software artifact generated for the course management system. This artifact is produced after the executable unified domain model has been created and deployed within the JDA framework, which transforms the model specifications into a functional application interface.



```
package org.jda.example.courseman.model;
```

Id: 33
Full name: Tran Van Anh

Tạo mới bản ghi dữ liệu: Student

Fig. 7. Software artifacts generated by the JDA framework

Discussion. We have successfully applied UDML in developing software for real-world problem domains. However, several validity concerns arise. First, correctness depends on the expressive adequacy of individual concern DSLs and their coherent integration via UDML, in line with DDD principles. Second, integrating heterogeneous external DSLs requires sufficient syntactic support, which is currently ensured through systematic metaconcept validation, interface design, transformation testing, and verification of generator rules. Third, the generated Java code may misalign with requirements if generation errors occur within the JDA framework. To mitigate this, we conduct careful review and verification of the generated output.

6 Related Work

We position our work at the intersection between the following areas: DSL Engineering, DDD concern DSLs, Integrating concern DSLs.

DSL Engineering and Language Workbenches. DSL engineering has matured significantly [34], with language workbenches such as JetBrains MPS [3] playing a pivotal role in enabling the design, implementation, and maintenance of DSLs for both academic and industrial use. The ability to define, extend, and compose DSLs in MPS has been demonstrated in various domains, including finance, healthcare, safety-critical systems, and industrial automation, where tailored DSLs have led to increased productivity, improved quality, and more effective leveraging of domain expertise.

DDD and Concern-Specific DSLs. DDD [10] advocates an iterative software development approach centered on a semantically rich domain model that encapsulates the core logic and rules of the problem space. Vernon [32] provides practical guidance for applying DDD principles in the construction of complex systems, while Jaiswal [17] addresses the gap between business requirements and object-oriented modeling in software development.

Several studies [2,4,11,29] emphasize maintaining strong consistency between design models and implementation source code, thereby enhancing correctness, maintainability, and extensibility. These works focus less on fully automating the development process or exhaustively modeling all system aspects, and more on ensuring reliable model–code alignment.

Valle [31] offers insights into the practical challenges software teams face in understanding and modeling complex business domains. Karagiannis [8] highlights domain-specific conceptual modeling as a crucial technique for producing high-quality information systems aligned with business needs. Similarly, Kapferer [18] leverages domain-specific languages (DSLs) and toolchains to standardize and automate aspects of Strategic DDD, improving both efficiency and design accuracy in complex software systems.

Wasowski [1] presents methods for designing, constructing, and applying DSLs to achieve effective modeling, automation, and reuse in software development. Voelter [33] shows how DSL engineering enables domain experts to directly express their concepts in source code, narrowing the gap between business requirements and technical implementation. Chiprianov [5] proposes the strategic integration of DSLs into the software engineering process to improve productivity, quality, and stakeholder communication.

Other works [6, 7, 21, 26] focus on applying DDD principles using internal DSLs—specifically annotation-based DSLs—embedded directly in object-oriented programming languages. This approach enables orthogonal integration, ensures model consistency across concerns, and facilitates formal reasoning and automated code generation. By leveraging projectional editing in JetBrains MPS and the DDD-oriented JDA framework, our approach bridges the gap between flexibility of internal DSLs and the rigor of external, metamodel-driven DSL engineering.

Integrating Concern DSLs. The work of Haber et al. [16] presents an effective solution for integrating heterogeneous modeling languages, enabling seamless interaction between distinct modeling paradigms. Nosal et al. [24] propose an innovative technique for combining heterogeneous programming languages and DSLs through the use of source code annotations. This approach enhances code expressiveness, simplifies the development and maintenance of complex systems, and allows developers to select the most suitable language for each task within the same source file. Loiret et al. [23] introduce a flexible method for incorporating domain-specific requirements into component-based systems using aspect-oriented programming (AOP) and annotations, effectively separating functional and non-functional concerns.

Pfeiffer [28] provides insights into effective strategies for constructing complex models through systematic composition. Similarly, Ugaz et al. [30] focus on methods and tools for weaving and integrating multiple aspects of Domain-Specific Modeling Languages, particularly cross-cutting concerns, in order to improve reusability, maintainability, and consistency in software systems.

Furthermore, several works [3, 14, 20, 27, 33] highlight JetBrains MPS as a powerful and flexible platform for DSL engineering. MPS facilitates the design, implementation, reuse, and composition of language components, enabling their seamless integration into existing software development workflows. This capability significantly improves the efficiency and quality of software engineering by promoting modularity and composability in language design.

In our previous work [22], we proposed an approach to composing concerns into an executable unified domain model. This approach leverages a metamodel based on UML metaconcepts to support the composition and unification of concern-specific DSLs. Each concern is specified using an external DSL, which is then mapped to an internal DSL to produce an executable domain model embedded in an object-oriented programming language (OOPL) for source code generation.

In contrast, the present work defines the full syntax of concern-specific DSLs and introduces an integration method based on tree merging at the AST level (Unified AST). Moreover, we define a unified syntax for composing concern-specific DSLs.

However, existing approaches often fall short in supporting the full DSL development pipelinespanning editing, verification, code generation, and automated testing. Backward traceability between models and generated code remains a critical challenge, especially as systems evolve.

7 Conclusion and Future Work

This paper introduces a methodology for composing heterogeneous concern DSLs into a unified domain model and defines the complete syntax and semantics of UDML to support domain-driven development. UDML is structured as an AST, where each node corresponds to a concept from a specific concern DSL. This structure enables modular development and seamless integration of heterogeneous concerns, while preserving consistency across structural, behavioral, and security dimensions of the domain model. The proposed approach is realized through a prototype tool developed using MPS and the JDA framework. The tool demonstrates the feasibility, expressiveness, modularity, and executability of UDML in modeling complex software systems aligned with DDD principles.

In the future, we plan to develop an MPS plug-in for our method for many domain concerns. We also intend to develop a technique for automatically transforming a specification requirement into complex software systems.

References

1. Andrzej Wąsowski, Thorsten Berger: Domain-Specific Languages Effective Modeling, Automation, and Reuse. Springer Cham (2023)
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan&Claypool, 2nd edn. (2017)
3. Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Domain-specific languages in practice: with JetBrains MPS. Springer Nature (2021)
4. Buede, D.M., Miller, W.D.: The Engineering Design of Systems: Models and Methods. Wiley, 4th edn. (2024)
5. Chiprianov, V., Kermarrec, Y., Rouvrais, S., Chiprianov, V., Kermarrec, Y., Rouvrais, S.: Integrating DSLs into a Software Engineering Process: Application to Collaborative Construction of Telecom Services. pp. 408–434. IGI Global (2013)
6. Dan Haywood: Apache Isis - Developing Domain-Driven Java Apps. Methods & Tools **21**(2), 40–59 (2013)
7. Dang, D.H., Le, D.M., Le, V.V.: AGL: Incorporating behavioral aspects into domain-driven design. Information and Soft.Tech **163**, 107284 (2023)
8. Dimitris, Karagiannis, Heinrich, C. Mayr, John, Mylopoulos: Domain-Specific Conceptual Modeling. Springer Cham (2016)
9. D.M. Le, D.-H. Dang, V.-H. Nguyen: Generative software module development for domain-driven design with annotation-based domain specific language. Information and Software Technology **120**, 106–239 (2022)
10. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional (2004)
11. Foster, E., Jr, B.T.: Software Engineering: A Methodical Approach, 2nd Edition. Auerbach Publications, New York, 2 edn. (Jul 2021)
12. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, 1st edn. (Sep 2010)
13. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE'07). pp. 37–54. IEEE (2007)
14. Fuksa, M., Sağlam, T., Neumann, T., Becker, S.: ALFI: Action Language for Foundational UML as an Intermediate Language for Model Transformations in JetBrains MPS. In: Proc. ACM/IEEE 27th Int. Conf. Model Driven Engineering Languages and Systems. pp. 1141–1145. MODELS Companion '24, Association for Computing Machinery, New York, NY, USA (Oct 2024). https://doi.org/10.1145/3652620.3688350, https://doi.org/10.1145/3652620.3688350
15. Gerasimov, A., Michael, J., Netz, L., Rumpe, B.: Agile Generator-Based GUI Modeling for Information Systems. In: Modelling to Program, vol. 1401, pp. 113–126. Springer International Publishing, Cham (2021)
16. Haber, A., Look, M., Perez, A.N., Nazari, P.M.S., Rumpe, B., Völkel, S., Wortmann, A.: Integration of heterogeneous modeling languages via extensible and composable language components. In: 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD). pp. 19–31. IEEE (2015)
17. Jaiswal, S.K., Agrawal, R.: Domain-Driven Design (DDD)- Bridging the Gap between Business Requirements and Object-Oriented Modeling. Int. Innovative Research in Engineering and Management **11**(2), 79–83 (2024), number: 2

18. Kapferer, S., Zimmermann, O.: Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling. In: Proc. 8th Int. Conf. Model-Driven Engineering and Software Development (MODELSWARD). vol. 1, pp. 299–306. SciTePress (2020)
19. Katara, M., Katz, S.: A concern architecture view for aspect-oriented software design. Software & Systems Modeling **6**(3), 247–265 (Aug 2007)
20. Latifaj, M., Taha, H., Ciccozzi, F., Cicchetti, A.: Cross-Platform Blended Modelling with JetBrains MPS and Eclipse Modeling Framework. In: ITNG 2022 19th Int. Conf. Information Technology-New Generations. pp. 3–10. Springer (2022)
21. Le, D.M., Dang, D.H., Nguyen, V.H.: On domain driven design using annotation-based domain specific language. Computer Languages, Systems & Structures **54**, 199–235 (Dec 2018)
22. Le, V.V., Dang, D.H.: An Approach to Composing Concerns for an Executable Unified Domain Model. In: Int. Conf. On Computing and Communication Technologies (RIVF). pp. 424–428 (Dec 2024), iSSN: 2473-0130
23. Loiret, F., Rouvoy, R., Seinturier, L., Romero, D., Sénéchal, K., Plsek, A.: An Aspect-Oriented Framework for Weaving Domain-Specific Concerns into Component-Based Systems. Journal of Universal Computer Science **17**(5), 742 (Mar 2011). https://doi.org/10.3217/jucs-017-05-0742
24. Nosál', M., Sulír, M., Juhár, J.: Language composition using source code annotations. Computer Science and Information Systems **13**(3), 707–729 (2016)
25. Ouchani, S., Debbabi, M.: Specification, verification, and quantification of security in model-based systems. Computing **97**(7), 691–711 (Jul 2015)
26. Paniza, J.: Learn OpenXava by example. CreateSpace, 1.1 edn. (2011)
27. Pech, V.: JetBrains MPS: Why Modern Language Workbenches Matter
28. Pfeiffer, J., Rumpe, B., Schmalzing, D., Wortmann, A.: Composition operators for modeling languages: A literature review. Journal of Computer Languages **76**, 101226 (2023), publisher: Elsevier
29. Suryn, W.: Software Quality Engineering: A Practitioner's Approach. John Wiley & Sons (Dec 2013), google-Books-ID: 1dNiAgAAQBAJ
30. Ugaz, R.: Weaving of Domain-Specific modelling Languages (2014), msdl.uantwerpen.be/people/rafael/mde/paper.pdf
31. Valle, I., Sales, T.P., Guerra, E., Daneva, M., Guizzardi, R., Bonino da Silva Santos, L.O., Proper, H.A., Guizzardi, G.: Unraveling the pain points of domain modeling. Information and Software Technology **183**, 107736 (Jul 2025). https://doi.org/10.1016/j.infsof.2025.107736
32. Vernon, V.: Implementing domain-driven design. Addison-Wesley Professional, 1 er edn. (2013)
33. Voelter, M.: Language and IDE Modularization and Composition with MPS. Lecture Notes in Computer Science (LNPSE), vol. 7680, pp. 383–430. Springer, Berlin, Heidelberg (2013)
34. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, implementing and using domain-specific languages. M Volter / DSLBook.org, Stuttgart, Germany (2013)
35. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C.: Self-optimizing AST interpreters. Proc. 8th symposium on Dynamic languages pp. 73–82 (Oct 2012). https://doi.org/10.1145/2384577.2384587