Information and Software Technology

AGL: Incorporating Behavioral Aspects into Domain-Driven Design

--Manuscript Draft--

Manuscript Number:	INFSOF-D-22-00490R2
Article Type:	Research paper
Keywords:	Domain-driven design (DDD); Module-based Architecture; Domain-specific language (DSL); UML/OCL-based domain modelling; Attribute-oriented Programming (AtOP)
Corresponding Author:	Duc-Hanh Dang VNU University of Engineering and Technology Department of Software Engineering Hanoi, VIET NAM
First Author:	Duc-Hanh Dang
Order of Authors:	Duc-Hanh Dang
	Duc Minh Le, Dr.
	Van-Vinh Le
Abstract:	Context: Domain-driven design (DDD) aims to iteratively develop software around a realistic domain model. Recent research in DDD has been focusing on using annotation-based domain-specific languages (aDSLs) to build the domain model. However, within current approaches behavioral aspects, that are often represented using UML Activity and State machine diagrams, are not explicitly captured in the domain model. Objective: The focus of this paper is to introduce a new approach for incorporating behavioral aspects into domain models within the Domain-Driven Design (DDD) approach. The proposed approach involves using a new activity graph language (AGL) as an aDSL for representing behavioral aspects within a unified domain model. This integration of AGL and the previously developed aDSL (DCSL to represent domain models) aims to achieve three important features of DDD: feasibility, productivity, and understandability. Method: Our approach involves building a unified class model in DCSL within a domain-driven architecture, which uses the annotation attachment feature of the host programming language (such as Java) to attach AGL activity graphs directly to the activity class of the unified class model, resulting in a unified domain model. In this work, we define the abstract and concrete syntax of AGL. To demonstrate our method, we use a Java framework called jDomainApp and evaluate AGL through a case study to show that it is expressive and practical for real-world software. Results: This paper presents two contributions. Firstly, it proposes a mechanism to include behavioral aspects in a unified domain model by introducing a new aDSL called AGL to represent domain behaviors. Secondly, it presents a unified modeling method for domain-driven software development. Conclusion: Our method significantly extends the state-of-the-art in DDD in two important fronts: constructing a unified domain model for both structural and behavioral aspects of domain models and bridging the gaps between model and code.

Department of Software Engineering
Faculty of Information Technology
VNU University of Engineering and Technology
144 Xuan Thuy, Cau Giay, Hanoi, VIETNAM

May 16, 2023

Prof. Miroslaw Staron
Editor-in-Chief
Information and Software Technology

Dear Prof. Miroslaw Staron,

We are pleased to submit our revised paper, “AGL: Incorporating Behavioral Aspects into Domain-Driven Design” (INFSOF-D-22-00490R1), for publication in Information and Software Technology. We appreciate the time and effort dedicated by the editorial staff and reviewers. The comments provided were valuable and helped us refine our paper. Below are our point-by-point responses to the reviewers’ comments.

Thank you very much!

Sincerely,

Duc-Hanh Dang

Response to Reviewers

General comments

The editor and the reviewers have examined the submission. Their general opinion is that after a revision addressing the reviewer's comments and after performing another round of peer evaluation, the manuscript could form a valuable contribution to the Journal.

1. Response to Reviewer #3

1.1. Summary

In this new version, the authors answered to most of my remarks.

- There are too many writing errors, making the paper unnecessarily hard to read.
- Sections 7 and 8 still need to improved, despite the changes.

1.2. Section 7

Q1. Section 7 is much better now, but it still lacks some technical details. Two of my questions remain unanswered:
- How annotations are inserted into Java or C# code?
- What is the input format?

Response: *We have made revisions to the paper in order to provide further clarity on the aspects related to the two questions. Firstly, we describe the specification of a unified domain model as the input of our method. This unified domain model, which is detailed in Section 7.2 (page 18), comprises a combination of the DCSL unified model and the AGL specification, both of which are encoded in Java. In our upcoming work, we intend to create transformations that allow us to derive the DCSL/AGL specifications from UML diagrams as input. Additionally, we acknowledge the need to improve these definitions for greater clarity: a unified class model (Definition 1, page 7), a DCSL unified model (Definition 2, page 8), and a unified domain model (Definition 7, page 15).*

Secondly, we elaborate in Section 7.2 (page 20, the last paragraph) on the implementation of AGL and its annotations in Java.

1.3. Section 8

Section 8 was also improved, I appreciate that the research questions are clearly stated. However, it is confusing at some points:
Q2. First, why do you separate the UML activity diagram from the UML class diagram? They represent the same model and are strongly connected to each other. Moreover, you say that "it lacks a mechanism to compose the behavioral diagrams with other structural diagrams", which is not true, since all UML diagrams are linked. Maybe I misunderstand the meaning of "mechanism to compose".

Response: *We concur with the reviewer's observation that there should be a connection and consistency between the UML Activity diagram and the UML Class diagram. It is important to note that all UML diagrams used to model the underlying system must be integrated and coherent. We have revised the underlying paragraph (page 22, line 8) to make it clear the point the reviewer mentioned: "Although the UML Activity diagram can be utilized to implement the final DDD pattern, it requires additional mechanisms such as fUML [34] and OCL [7] to obtain an integrated semantic model of the behavioral and structural diagrams (one of which represents the domain model)."*

Q3. Second, Table 3B should be better explained. For instance, you explain that ApacheIsis supports 4 out of 8 properties of the domain field pattern, but you do not explain these properties and neither why ApacheIsis cannot represent them. You should provide examples of each language to illustrate your claims.

Response: *We have revised the paper to provide a more detailed explanation of Table 3B. Specifically, as the reviewer's suggestion, we have listed the essential properties w.r.t. the meta-attributes DAttr and DAssoc in order to clarify the values presented in Table 3B. Due to the limited space of this paper, examples for each language to illustrate our claims, as well as a detailed explanation of the annotations of the languages ApacheIsis[1] and OpenXava[2] are provided in the accompanying technical report [27].*

1.4. Other remarks

In section 2.4, you explain that you need a mechanism "to maintain consistency between the two models". Which models?

Response: *We have revised the paragraph in question (page 6, line 7) to clarify the aforementioned point: "To create an executable version of the software, we need to integrate domain behavior with the essential domain model depicted in Figure 3. Currently, this domain*

[1] https://svn.apache.org/repos/infra/websites/production/isis/content/guides/rg.html
[2] https://www.openxava.org/OpenXavaDoc/docs/annotations_en.html

behavior is represented in UML, which requires us to adopt a mechanism for ensuring consistency between the behavioral model and the essential domain model, typically at the implementation level."

In section 6.1, you talk about "UML activity graph requirements". Could you be more precise about this requirements? In this same section, you explain that "variable is an alternative to object flows". Could you explain?

Response: *We have addressed the reviewer's points by updating Section 6.1 (page 16). We have clarified that the term "UML activity graph requirements" refers specifically to the requirements for Activity as explained in [5, p. 373]. The revision is as follows:*

"We define the AGL's domain requirements by applying inclusion (I), exclusion (X), and restriction (R) clauses to the UML activity requirements as detailed in [5, p. 373]. Specifically, the following clauses apply:

- *(I1) a module action, as discussed in Section 4, is a special form of action [5, p. 441];*
- *(R1) each executable node [5, p. 403] performs a sequence of module actions;*
- *(X1) using a variable with activity [5, p. 377];*
- *(X2) using variable actions [5, p. 469].*

I1 and R1 are necessary to integrate the activity graph into MOSA. X1 and X2 exclude variable usage, which is an alternative to object flow, the primary means for moving data in UML activities [5, p. 377]. The AGL activity model captures current system state by directly referencing the unified class model, instead of using object flow."

Note that clauses X3 (which excluded guards from activity edges [5, p. 373]) and R2 (which restricted value specification [5, p. 374] to decision nodes) have been removed in this revised version of the paper. In the previous version, we had intended to apply these clauses so that the guard condition of a decision node could be checked by a method of a Java abstract class called "Decision". However, we have since found that this approach is not necessary, and have therefore removed these clauses.

== Minor remarks

- "work" is an uncountable noun, like "software" or "information".

=== Introduction

- "which both thoroughly captures" -> "which thoroughly captures"
- "Recent works in DDD [2, 3] proposed annotation-based"
  -> "Recent work in DDD [2, 3] propose annotation-based"
- "We aim to define an extension of domain model"
  -> "We aim to define an extension of the domain model"
- " the software at higher level" -> " the software at a higher level"

- "As a first step to get over this point is we define"
  -> "As a first step to get over this point, we define"
- "with a language support" -> "with language support"
- "behavior aspects" -> "behavioral aspects" (several times)

=== Section 2

- "Two main features of DDD is that (1)" -> "The two main features of DDD are: (1)"
- "that are expressed in a so-called the ubiquitous language"
  -> "that are expressed in a ubiquitous language"
- "Our previous works [8, 15] proposed a variant"
  -> "Our previous work [8, 15] propose a variant"
- "Meta-concept Associative Field represents Domain Field"
  -> "Meta-concept Associative Field represents the Domain Field"
- "Finally, meta-concept Domain Method is composed of Method and"
  -> "Finally, the meta-concept Domain Method is composed of a Method and"
- "to maintain a consistency between" -> "to maintain consistency between"
- "Figure 3 shows an essential domain model for CourseMan, that is represented by a UML class diagram"
  -> "Figure 3 shows an essential domain model for CourseMan, which is represented by a UML class diagram"

=== Section 3

- "First, we take as input domain requirements that are captured by" -> "First, we take as input domain requirements which are captured by"
- "We then aim to represent such input domain requirements as a composition of a DCSL model for a so-called unified model and an AGL model to represent domain behaviors."
  -> This sentence should be rephrased
- "For the latter one (the AGL model)," -> "For the AGL model,"
- "referred to as a so-called activity class" -> "referred to as an activity class"
- "(coordinating a collaboration among moudules)"
  -> "(coordinating collaboration among modules)"
- "Domain behaviors are specified using UML Activity diagram"
  -> "Domain behaviors are specified using the UML Activity diagram"

=== Section 4

- "characterised" -> "characterized"
- "This ASE consists in a sequence" -> " This ASE consists of a sequence"
- "so that interested listeners of this event can handle"
  -> "so that interested listeners of this event can handle it"

=== Section 5

- "that could be captured at a high-level description using an UML activity diagram together with domain-model based statements"

-> "that could be captured at a high-level description using a UML activity diagram together with domain-model based statements"

- "The first catalog of domain behavior patterns is defined corresponding to"
  -> "The first catalog of domain behavior patterns is defined as corresponding to"
- "For brevity, we will omit" -> "For brevity, we omit"
- "We would illustrate each pattern" -> "We illustrate each pattern"
- "The third and fourth ANodes represent the two decision cases: the first results in creating a new C1 object for the specified Cd object, the second,"
  -> "The third and fourth ANodes represent the two decision cases: the first results in creating a new C1 object for the specified Cd object, and the second,"
- "It uses two variables k and kout, both are dependent on Ck. "
  -> ". It uses two variables k and kout, which are both dependent on Ck. "
- “To obtain a concrete AGL specification when applying a domain behavior pattern, basically, we proceed three main steps as follows:”
  -> “To obtain a concrete AGL specification when applying a domain behavior pattern, we follow three main steps:”
- "The reference from ANodes to domain classes (expressed with the keywords refCls and outCles) provide"
  -> "The reference from ANodes to domain classes (expressed with the keywords refCls and outCles) provides"
- "keywork" -> "keyword"
- "of an activity of the domain" -> "of a domain activity"

=== Section 6

- "This section briefly specify" -> "This section briefly specifies"
- "(R2) value specification [5, p. 374]) is only applied to decision node;"
  -> "(R2) value specification [5, p. 374]) is only applied to decision nodes;"
- "(X1) using variable with activity ([5, p. 417]); (X2) variable action [5, p. 467]; (X3) activity edge [5, p. 373] is without guards."
  -> Please rephrase using articles
- "According to the UML specification, variable is an alternative to using object flow."
  -> "According to the UML specification, a variable is an alternative to using object flow. "
- "In this figure the entire AGL specification" -> "In this figure, the entire AGL specification"
- "Attribute label realizes the node label." -> "Attribute label represents the node label."

=== Section 7

- "with a focus on explaining main design" -> "with a focus on explaining the main design"
- "In order to obtain an OrderMan software" -> "To obtain the OrderMan software "
- "jdomainapp" -> "Jdomainapp"
- "java" -> "Java"
- "organising" -> "organizing"

=== Section 8

- "as a domain-specifying language" -> "as a domain-specific language"
- "expressiveness: the extend" -> "expressiveness: the extent"
- "constructiability: the extend" -> "constructibility: the extent"
- "in an piecewise" -> "in a piecewise"
- "perform on representing" -> "perform in representing" (2x)
- "How much effort is required to define unified domain model in AGL+ for generating a DDD software?"
  -> "How much effort is required to define a unified domain model in AGL+ for generating DDD software?" (2x)
- "based on UML Class diagram" -> "based on the UML Class diagram"
- "The last pattern could be realized by UML Activity diagram"
  -> "The last pattern could be represented in the UML Activity diagram"
- "AGL's RCL" -> did you mean "RLC" ?

=== Section 9

- "that were discussed in Section 11" -> "that is discussed in Section 11"
- "specifciation" -> "specification"

Response: *Thank you for your suggestions. We have updated the paper with the suggestions. (Specifically, for the "AGL's RCL" we fix it with "RCL" for "Required Coding Level".)*

2. Response to Reviewer #4

2.1. Summary

The authors have greatly improved their paper compared to the initial version. I am satisfied with the changes and recommend accepting the paper.

2.2. Comment

The only relevant issue is that I could not access the accompanying technical report at https://tinyurl.com/AGLTechnical, the authors should make sure that the provided URL is correct. A final proofreading is also recommended to fix the (few) remaining language issues.

Response: *We have fixed the issue so that the accompanying technical report is accessible with the provided URL. We have also carefully revised the paper to fix the remaining language issues.*

2.3. Other remarks

- Page 2: to get over this point is we ---- to get over this point, we
- In Figure 2, attribute Student.name is assigned optional=false, but the text mentions that this attribute is optional. It is a bit confusing.
- Page 8: an unified ---- a unified
- Page 20: the extend ---- the extent
- Page 20: constructiability ---- constructability
- Page 20: an piecewise ---- a piecewise
- In the first paragraph of Section 9, do not mention reliability, as it is not discussed in the subsequent subsections.
- Page 24: lead to unsatisfactory. ---- lead to an unsatisfactory model.
- Page 24: allowing combined the class model ---- allowing combining the class model
- Page 24: specification ---- specification
- Page 26: to incorporates ---- to incorporate

Response: *Thank you for your suggestions. We have updated the paper with the suggestions.*

Structured Abstract

AGL: Incorporating Behavioral Aspects into Domain-Driven Design

Duc-Hanh Dang[a,b,*], Duc Minh Le[c], Van-Vinh Le[a]

[a] *Department of Software Engineering, VNU University of Engineering and Technology, Vietnam*
[b] *Vietnam National University, Hanoi*
[c] *Department of Information Technology, Swinburne Vietnam, FPT Univeristy*

Abstract

Context: Domain-driven design (DDD) aims to iteratively develop software around a realistic domain model. Recent research in DDD has been focusing on using annotation-based domain-specific languages (aDSLs) to build the domain model. However, within current approaches behavioral aspects, that are often represented using UML Activity and State machine diagrams, are not explicitly captured in the domain model.
Objective: The focus of this paper is to introduce a new approach for incorporating behavioral aspects into domain models within the Domain-Driven Design (DDD) approach. The proposed approach involves using a new activity graph language (AGL) as an aDSL for representing behavioral aspects within a unified domain model. This integration of AGL and the previously developed aDSL (DCSL to represent domain models) aims to achieve three important features of DDD: feasibility, productivity, and understandability.
Method: Our approach involves building a unified class model in DCSL within a domain-driven architecture, which uses the annotation attachment feature of the host programming language (such as Java) to attach AGL activity graphs directly to the activity class of the unified class model, resulting in a unified domain model. In this work, we define the abstract and concrete syntax of AGL. To demonstrate our method, we use a Java framework called jDomainApp and evaluate AGL through a case study to show that it is expressive and practical for real-world software.
Results: This paper presents two contributions. Firstly, it proposes a mechanism to include behavioral aspects in a unified domain model by introducing a new aDSL called AGL to represent domain behaviors. Secondly, it presents a unified modeling method for domain-driven software development.
Conclusion: Our method significantly extends the state-of-the-art in DDD in two important fronts: constructing a unified domain model for both structural and behavioral aspects of domain models and bridging the gaps between model and code.

Keywords: Domain-driven design (DDD); Module-based Architecture; Domain-specific language (DSL); UML/OCL-based domain modelling; Attribute-oriented Programming (AtOP)

*Corresponding author
Email addresses: hanhdd@vnu.edu.vn (Duc-Hanh Dang), duclm20@fe.edu.vn (Duc Minh Le), vinhskv@gmail.com (Van-Vinh Le)

AGL: Incorporating Behavioral Aspects into Domain-Driven Design

Duc-Hanh Dang[a,b,*], Duc Minh Le[c], Van-Vinh Le[a,b,d]

[a] *Department of Software Engineering, VNU University of Engineering and Technology, Vietnam*
[b] *Vietnam National University, Hanoi*
[c] *Department of Information Technology, Swinburne Vietnam, FPT Univeristy*
[d] *Department of Information Technology, Vinh University of Technology Education, Vietnam*

Abstract

Context: Domain-driven design (DDD) aims to iteratively develop software around a realistic domain model. Recent research in DDD has been focusing on using annotation-based domain-specific languages (aDSLs) to build the domain model. However, within current approaches behavioral aspects, that are often represented using UML Activity and State machine diagrams, are not explicitly captured in the domain model.
Objective: The focus of this paper is to introduce a new approach for incorporating behavioral aspects into domain models within the Domain-Driven Design (DDD) approach. The proposed approach involves using a new activity graph language (AGL) as an aDSL for representing behavioral aspects within a unified domain model. This integration of AGL and the previously developed aDSL (DCSL to represent domain models) aims to achieve three important features of DDD: feasibility, productivity, and understandability.
Method: Our approach involves building a unified class model in DCSL within a domain-driven architecture, which uses the annotation attachment feature of the host programming language (such as Java) to attach AGL activity graphs directly to the activity class of the unified class model, resulting in a unified domain model. In this work, we define the abstract and concrete syntax of AGL. To demonstrate our method, we use a Java framework called jDomainApp and evaluate AGL through a case study to show that it is expressive and practical for real-world software.
Results: This paper presents two contributions. Firstly, it proposes a mechanism to include behavioral aspects in a unified domain model by introducing a new aDSL called AGL to represent domain behaviors. Secondly, it presents a unified modeling method for domain-driven software development.
Conclusion: Our method significantly extends the state-of-the-art in DDD in two important fronts: constructing a unified domain model for both structural and behavioral aspects of domain models and bridging the gaps between model and code.

Keywords: Domain-driven design (DDD); Module-based Architecture; Domain-specific language (DSL); UML/OCL-based domain modelling; Attribute-oriented Programming (AtOP)

1. Introduction

The object-oriented domain-driven design (DDD) [1] is aimed at developing software in an iterative manner around a realistic model of the problem domain, which captures the domain requirements and is technically feasible for implementation. To achieve this, a close collaboration among all stakeholders, including domain experts, end-users, and developers, is required. The ubiquitous language [1] is used to construct the domain model and create an object-oriented implementation of this model. The DDD method employs a conceptual layered software architecture, which places the domain model at the core layer and

*Corresponding author
Email addresses: hanhdd@vnu.edu.vn (Duc-Hanh Dang), duclm20@fe.edu.vn (Duc Minh Le), 21028005@vnu.edu.vn (Van-Vinh Le)

other architectural concerns, such as user interface, persistence, etc., in other layers surrounding this core. Recent research in DDD [2, 3] proposes annotation-based domain-specific languages (aDSLs), which are written inside a host object-oriented programming language (OOPL), to facilitate the construction of domain models. A straightforward way to obtain an executable version of the software from such a representation of the domain model is to directly embed the implementation in the OOPL along with other concerns for the entire program. Another indirect way is to follow model-driven approaches by composing the domain model with other concerns expressed at a high level using a general language like UML and DSLs. The final program can then be obtained by model transformations, either model-to-model or model-to-text. This work focuses on an alternative approach to achieve this goal, which is a refinement of an aDSL-based software development method for DDD that we proposed in a recent work [4].

We aim to develop an extension of the domain model that can capture behavioral aspects of the domain, resulting in a unified domain model that facilitates software construction. However, modeling behavioral aspects within this unified domain model can be challenging. To overcome this challenge, we propose a language with specific support for incorporating domain behaviors. This language aims to bridge the gap between the domain model and its implementation, making it easier to build software through model transformations from a domain model that incorporates behavioral models expressed in UML and DSLs.

The proposed language, AGL (Activity Graph Language), has two main aims: (1) to represent behavioral aspects that could be captured using UML Activity diagrams and Statecharts [5], and (2) to incorporate these aspects as part of the unified domain model. AGL is scoped around a restricted domain of the UML activity language that is based on essential UML activity modeling patterns [5, p. 373]. The meta-modeling approach for DSLs [6] is used, with UML/OCL [5, 7] to specify the abstract and concrete syntax models of AGL. To incorporate AGL into the unified domain model, the previously-developed aDSL, DCSL [4], is used to express a *unified class model*. The unified class model is viewed as an extended domain model in MOSA (a module-based software architecture recently developed for DDD [8]) and includes new domain classes, referred to as *activity classes*, that are attached with AGL's activity graph. Each activity class corresponds to an executable node of AGL's activity graph, which performs a set of core actions on the software modules in MOSA. The composition of the unified class model and the AGL specifications results in a *unified domain model*, which is essentially expressive and usable for designing real-world software. The method is implemented in jDomainApp and evaluated to demonstrate its effectiveness.

In brief, our paper makes the following contributions:

- A mechanism to incorporate behavioral aspects for a unified domain model: An aDSL (named AGL) is defined to represent the domain behaviors for the incorporation;
- A unified modeling method for domain-driven software development;
- An implementation in the jDomainApp framework for the proposed method; and
- An evaluation of AGL to show that it is essentially expressive and usable for designing real-world software

The rest of the paper is structured as follows. Section 2 presents our motivating example and the technical background. Section 3 overviews our approach to incorporating behavioral aspects into a domain model. Section 4 provides formal semantics for module actions. Section 5 explains the patterns to capture domain behaviors. Section 6 specifies AGL. Section 7 discusses the case study OrderMan and tool support. An evaluation of AGL is presented in Section 8. Section 9 discusses threats to the validity of our work. Section 10 reviews the related work. This paper closes with a conclusion and an outlook on future work.

2. Motivating Example and Background

This section motivates our work through an example and reviews the background concepts that form the basis for our discussion in this paper.

2.1. A Brief Overview of Domain-Driven Design (DDD)

Domain-driven design (DDD) [1] aims to iteratively develop software around a realistic model of the application domain, which on the one hand thoroughly captures the domain requirements. On the other hand, the model is technically feasible for implementation. According to Evans [1], OOPLs such as Java are a natural fit for use with DDD. Booch [9] had earlier pointed out domain models in OOPL should be expressive and feasible because of two main points. First, object naturally represents entities that exist in real-world domains. Second, the construct of object used in OOPL is also a basic construct of modeling languages for high-level analysis and design, that conceptualize and realize the domain. This work uses DDD to refer specifically to object-oriented DDD. As explained in [1], within the DDD approach domain model tends to be the heart of software, which is where the complexity lies. Two main features of DDD are: (1) feasibility, i.e., a domain model should be the code and vice versa, and (2) satisfiability, i.e., the domain model would satisfy the domain requirements that are expressed in a ubiquitous language [1]. This language is defined for stakeholders, including the domain experts and developers, in an iterative and agile process of eliciting the domain requirements. To obtain these two main features of DDD can be seen as one of the main focus of current research on DDD.

2.2. MOSA: A Module-Based Software Architecture for DDD

The adoption of the Model-View-Controller (MVC) architecture in practical software development is common, especially for software that requires a graphical user interface (GUI) to assist the development team. The reason for this is due to the belief that software construction cannot be fully automated because of the human factors involved in the development process [10]. The MVC architecture consists of three components: model, view, and controller, and each component's internal design is maintained independently with minimal impact on the other two components. Modularity can be further enhanced by applying the architecture at the module level, creating a hierarchical design architecture in which software is composed of a hierarchy of software modules. At the module level, another agent-based design architecture named PAC [11] can be adopted to realize a coherent subset of the software's functions in terms of the architectural components. These software modules are also referred to as agents in general [12].

To construct DDD software from a domain model, an architectural model that follows the generic layered architecture [1, 13] is necessary. This model places the domain model at the core layer and separates it from the user interface and other layers. The MVC architecture model [14] is one example of such a model, according to Evans [1]. Existing DDD frameworks [2, 3] also use the MVC architecture. The user interface is important in presenting the domain model to stakeholders and helping them to build the model effectively. Therefore, we believe that any DDD tool that follows the DDD's layered architecture must be based on the MVC architecture.

In our previous work [8, 15], we proposed a variant of the MVC architecture called **module-based software architecture** (**MOSA**) for DDD software. MOSA supports the automatic generation of software modules from the domain model, as well as the software generated from these modules. A **MOSA model** consists of a set of MVC-based **module classes**, each of which is an MVC-based structured class [5] representing a module with three components: a domain class (the model), a view class (the view), and a controller class (the controller). The module class takes ownership of the model, view, and controller. The view and controller are parameterized classes created by binding the template parameters of two library template classes, named `View` and `Controller` (*resp.*), to the domain class. In our previous work [8], we presented a technique for semi-automatically generating a module class from the domain class it owns. Moreover, the view is designed to reflect the model structure. A set of module classes is used as input for the jDomainApp software framework [16] to automatically generate software. We assume that a module class is defined for every domain class in this paper.

Example. Figure 1 illustrates five module classes belonging to the course management domain (CourseMan). The dashed lines in the figure represent parameter bindings, with the ends of the `Controller` and `View` components marked by the symbol '◯'. For example, the module class `ModuleStudent` consists of three component classes: the domain class `Student`, the view `View`⟨`Student`⟩, and the controller `Controller`⟨`Student`⟩.

We argue that MOSA captures the essence of object-oriented software design through a modular, MVC-based structure. According to Booch [9], object-oriented software consists of objects and their interactions,

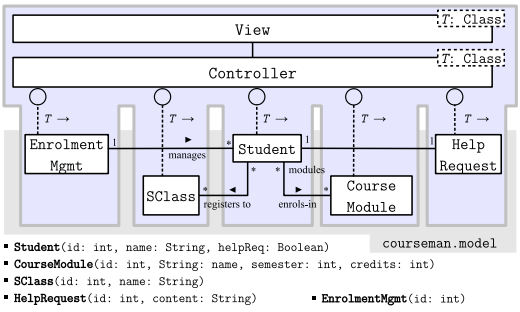


Figure 1: The MOSA model of CourseMan.

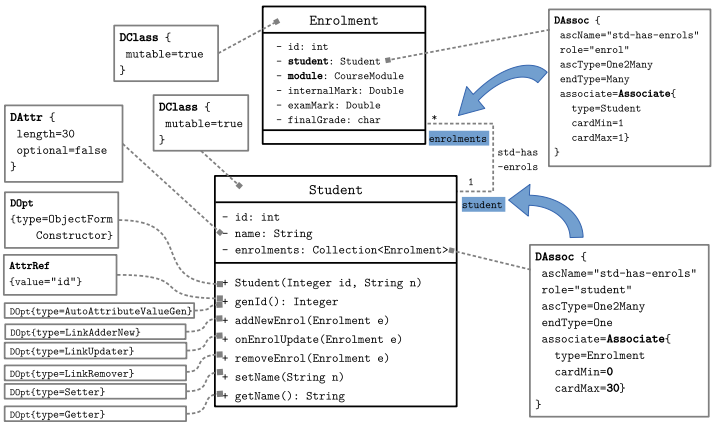


Figure 2: A partial CourseMan domain model in DCSL [4].

which are realized through behavioral invocation. Since the domain model is expressed in DCSL (further explained in Section 2.3), the MOSA model, with the domain model at its core, helps produce software with essential behaviors. First, objects are instances of the domain classes represented in DCSL with essential structural features. Second, interaction among objects in a group of domain classes is performed through an event-based message passing mechanism managed by the owner modules of those classes. This mechanism, as described in detail in [16], maps events to essential behaviors supported in DCSL. Events can be triggered by user interactions on a view of a concerned module.

2.3. Representing Domain Models in DCSL

In the previous work [4] we have defined an *annotation-based domain specific language* (aDSL) named *Domain class specification language* (DCSL) in order to express the domain models.

Annotation-Based Domain Specific Language (**aDSL**) is coined in [17] as an attempt to formalise the notion of fragmentary, internal DSL [18] for the use of annotation to define DSLs. An aDSL is defined based on a set of meta-concepts that are common to two popular host OOPLs, Java [19] and C# [20], as described in [4]. These meta-concepts include class, field, method, parameter, annotation, and property. Our idea of using annotation to represent modeling rules and constraints is inspired by AtOP [21–24], which extends a conventional program with a set of attributes that capture application- or domain-specific semantics [22]. These attributes are represented in contemporary OOPLs as annotations. We identified three important benefits of using aDSL for DDD: feasibility, productivity, and understandability, as described in our previous work [4]. Feasibility comes from the fact that the domain model is feasible for implementation in a host OOPL. Productivity is achieved by leveraging the host language platform tools and libraries to process and transform the domain model into other forms suitable for constructing the software. Understandability of the domain model code is enhanced with the introduction of domain-specific annotations.

Domain class specification language (**DCSL**) [4] is a horizontal aDSL developed by us to express domain models. A key feature of DCSL is that its meta-concepts model the domain-specific terms composed of the core OOPL meta-concepts and constraints. More specifically, the meta-concept **Domain Class** is composed of the meta-concept `Class` and a constraint captured by an annotation named `DClass`, which states whether or not the class is mutable. Similarly, the meta-concept **Domain Field** is composed of the meta-concept `Field` with a set of state space constraints, represented by an annotation named `DAttr`. The meta-concept `AssociativeField` represents the **Domain Field** that realizes one end of the association between two domain classes. DCSL supports all three types of association: one-to-one (abbr. one-one), one-to-many (abbr. one-many), and many-to-many (abbr. many-many). Finally, the meta-concept **Domain Method** is composed of `Method` and commonly used constraints and behavioral types that are imposed on instances of domain classes. The essential behavioral types are represented by an annotation named `DOpt` and another annotation named `AttrRef`. The latter references the domain field that is the primary subject of a method's behavior.

Example. Figure 2 shows a partial CourseMan's domain model expressed in DCSL. This model involves two domain classes: `Student` and `Enrolment`. Both of them are assigned with a `DClass` element, which states that they are mutable domain classes (`DClass.mutable` = `true`). In particular, class `Student` has three domain fields: `id`, `name`, and `enrolments`. Domain field `Student.name` is illustrated with an `DAttr` element which states that it is not an optional domain field (`DAttr.optional` = `false`), whose maximum length is 30 characters (`DAttr.length` = 30). An optional domain field means that the value of this field needs not be initialised when an object is created. Domain field `Student.enrolments` is an associative field, which is assigned with a `DAssoc` element. This element specifies the `Student`'s end of the association with `Enrolment`. The opposite end of this association is specified by another `DAssoc` element that is assigned to the associative field `Enrolment.student`. The two thick arrows in the figure map the two `DAssoc` elements to the two association ends. The seven methods of class `Student` listed in the figure are domain methods. Each method is assigned with a `DOpt` element, which specifies the behavioral type. For instance, method `genId`, whose behavioral type is `AutoAttributeValueGen`, is additionally assigned with an `AttrRef` element, which references the name of the domain field `Student.id`. This means that `genId` is the method that automatically generates values for `Student.id`.

2.4. Motivating Example and Research Questions

We adapt a compact and essential software domain from a previous work [4], named course management domain (CourseMan) as our motivating example.

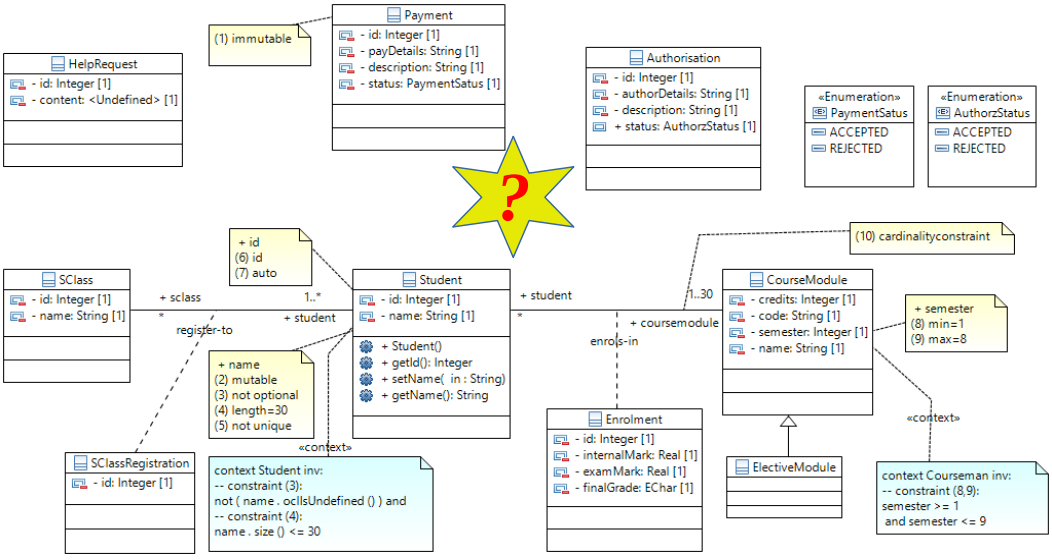


Figure 3: The essential domain model of CourseMan.

Figure 3 shows an essential domain model for CourseMan, which is represented by a UML class diagram together with OCL constraints. Within our DDD approach [4] this domain model would be represented in DCSL. As shown in the bottom part of the figure, this domain model includes four main classes and two association classes: Class `Student` represents students that register to study in an academic institution; Class `CourseModule` represents the course modules that are offered by the institution; Class `ElectiveModule` represents a specialized type of `CourseModule`; Class `SClass` represents the student class type for students to choose; Association class `SClassRegistration` captures details about the many-many association between `Student` and `SClass`; and association class `Enrolment` captures details about the many-many association between `Student` and `CourseModule`. As shown in the top part of the figure (with a star-like shape labeled "?"), this domain model includes also three other classes captured for an enrolment management activity:

- `HelpRequest`: captures data about help information provided to students.
- `Payment`: captures data about payment for the intuition fee that a student needs to make.
- `Authorisation`: captures data about the decision made by an enrolment officer concerning whether or not to allow a student to undertake the registered course modules.

Figure 4 depicts a UML Activity diagram for the enrolment management activity, which entails registering `Students`, enrolling them in `CourseModules`, and registering them in `SClasses`.

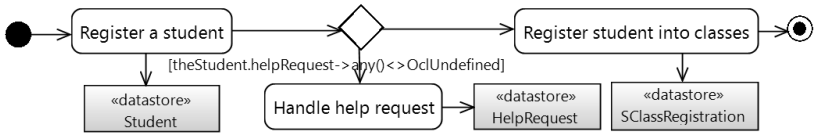


Figure 4: A UML Activity diagram to represent the enrolment management activity.

Additionally, it allows `Students` to request help during the enrolment process. To create an executable version of the software, we need to integrate domain behavior with the essential domain model depicted in Figure 3. Currently, this domain behavior is represented in UML, which requires us to adopt a mechanism for ensuring consistency between the behavioral model and the essential domain model, typically at the implementation level. As an alternative approach to addressing this challenge, we propose to treat the behavior concern as an extension of the essential domain model for a unified domain model. This follows the DDD approach that we introduced in our previous work [4]. The unified domain model embodies key features such as feasibility, productivity, and understandability, as explained previously. Achieving this goal poses two primary challenges that motivate our work:

1. How can we incorporate domain behaviors (that can be captured by UML Activity diagrams) into a domain model for a composition of both structural and behavioral aspects of the domain?

2. How can we extend a domain definition language like DCSL with new constructs to represent such domain behaviors?

3. Overview of the Proposed Approach

This section presents our approach to incorporating behavioral aspects as part of a unified domain model. Figure 5 provides an overview of our proposed method, which involves three iterative steps. First, we start with domain requirements that are represented by an essential domain model, which captures the structural view with domain concepts, and UML Activity diagrams, which represent domain behaviors. We aim to combine these input domain requirements into a *unified domain model*, which is composed of a DCSL model and an AGL model. The DCSL model extends the essential domain model to relate the structure view with the behavioral view. The AGL model represents domain behaviors. A more detailed definition of the DCSL model is provided in Section 3.2. Sections 3.1 and 4 explain in detail how we define the AGL model. Second, we take the unified domain model as input to automatically generate GUI- and module-based software. The generated software is presented to the domain expert for feedback. Third, if there is feedback, we update the input model and continue the cycle. If the domain expert is satisfied with the models, the cycle ends.

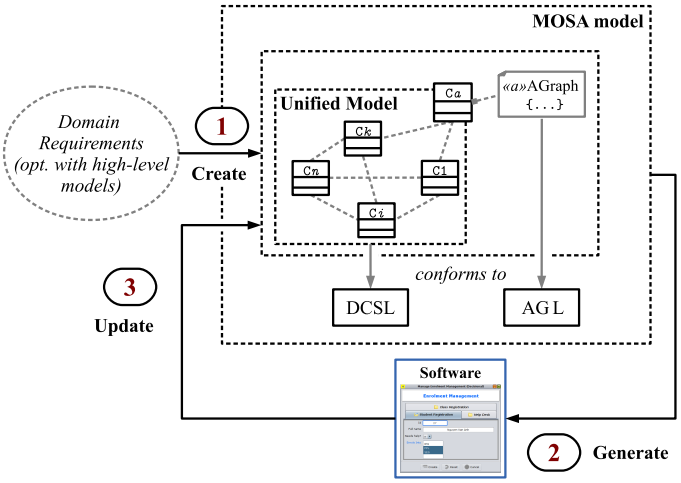


Figure 5: An overview of our method.

3.1. Incorporating Domain Behaviors

Our proposed mechanism for incorporating domain behaviors into a domain model is based on the structure and behavioral semantics of MOSA at two points. Firstly, we define a set of essential actions for each module class that owns a corresponding domain class. These essential actions are *atomic actions* (further explained in Section 4), that enable manipulation of domain class instances. Secondly, domain behaviors are viewed as collaborations among modules in MOSA. Each module collaboration is coordinated

by a composite module and captured by a corresponding activity model. We map each activity model to a new domain class, known as an activity class, which is owned by a corresponding activity module. For instance, the enrolment management activity in CourseMan (as shown in Figure 4) is mapped to an activity class owned by the `ModuleEnrolmentMgmt` in CourseMan. The containment tree of the composite module allows us to promote it as the main module for managing the entire activity.

The proposed mechanism allows for the use of UML Activity diagrams to represent domain behaviors, but with restrictions to ensure that the diagrams correspond to the behavioral semantics of the composite module that coordinates collaboration among modules. To achieve this, a pattern-based approach is employed where domain behaviors are specified using UML Activity diagrams with basic constructs that correspond to the five essential activity modeling patterns presented in [4]. These patterns are named after the five elementary activity flows: *sequential*, *decisional*, *forked*, *joined*, and *merged*. A more detailed explanation of this approach is provided in Section 5.

3.2. Unified Class Model

A *unified class model* is an extension of the domain model that enables the incorporation of domain behaviors. In our approach, domain behaviors are captured as activity models using UML Activity diagrams. We add *activity classes*, such as class $\texttt{C}_\texttt{a}$ shown in Figure 5, to represent each activity of the domain behaviors. These activity classes are associated with corresponding activity models, which act as activity graphs and synchronize the behavioral logic of the activity with the current states of the domain model. The resulting unified class model can be implemented in DCSL, and we refer to it as a DCSL unified model.

Definition 1. *Let an activity model be specified using a UML activity diagram for domain behaviors. A* ***unified class model*** w.r.t. *the activity model is a domain model extended with the following features:*

- ***activity class****: a domain class that represents the activity.*
- ***data component class*** *(or* ***data class*** *for short): a domain class that represents each data store.*
- ***control component class*** *(or* ***control class****): captures the domain-specific state of a control node. A control class that represents (does not represent) a control node is named after (the negation of) the node type; e.g., , decision (non-decision) class, join (non-join) class,* etc.
- ***activity-specific association****: an association between each of the following class pairs:*
  - *activity class and a merge class.*
  - *activity class and a fork class.*
  - *a merge (fork) class and a data class that represents the data store of an action node connected to the merge (fork) node.*
  - *activity class and a data class that does not represent the data store of an action node connected to either a merge or fork node.*

We will collectively refer to the data and control classes of an activity class model as ***component classes****.* □

Note that the representation scheme in the above definition does not cover *all* the possible associations among the component classes. It focuses only on the activity-specific ones, i.e., in general, we just focus on a restricted semantic domain of UML Activity diagrams for AGL. These associations play two important roles. First, they explicitly model the links between domain-specific states of the activity nodes. Second, they are used to incorporate the modules of the data and control classes into the containment tree of the activity module, thereby promoting this module as the main module for managing the entire activity.

The condition imposed on the fourth class pair of activity-specific association stems from the fact that there is no need to explicitly define the association between an activity class and a data class that represents the data store of an action node connected to either a merge or fork node. Such a data class is 'indirectly' associated to the activity class, via two associations: one is between it and the merge or fork class (the third class pair), and the other is between the activity class and this control class (the first or second class pair).

Definition 2. *A **DCSL unified model** is a DCSL model that realizes a unified class model as follows:*

- *a domain class c_a (called the **activity domain class**) to realize the activity class.*
- *the domain classes $c_1, \ldots, c_n$ to realise the component classes.*
- *let $c_{i_1}, \ldots, c_{i_k} \in \{c_1, \ldots, c_n\}$ realize the non-decision and non-join component classes, then $c_a, c_{i_1}, \ldots, c_{i_k}$ contain associative fields that realize the corresponding association ends of the relevant activity-specific associations.* □

In the remainder of this paper, to ease notation we will use **activity class** to refer to the activity domain class c_a and **component class** to refer to the $c_1, \ldots, c_n$.

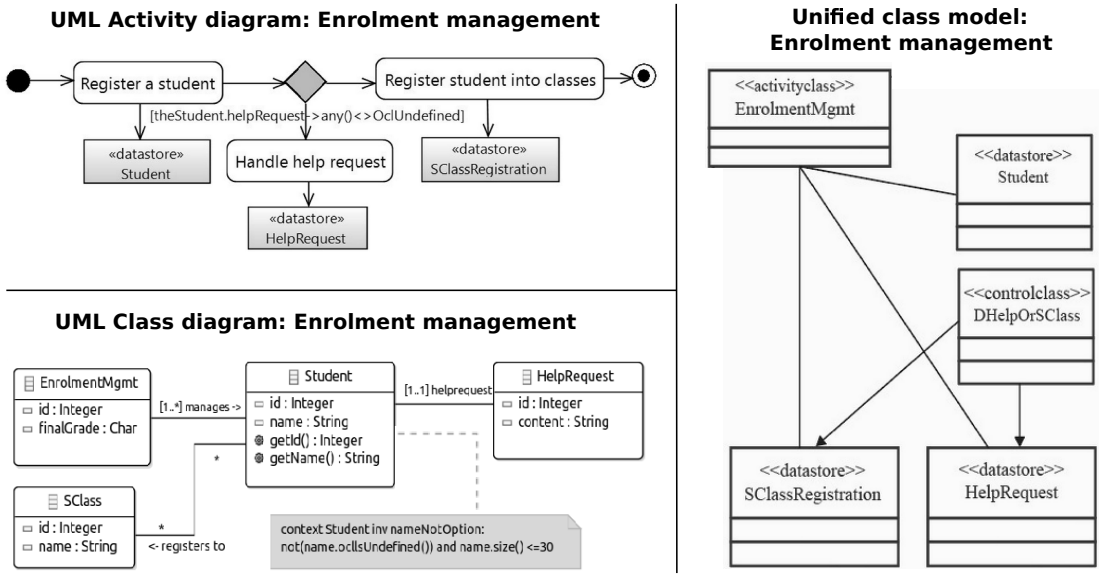


Figure 6: (A: Left) The UML activity and class diagrams of a CourseMan software variant that handles the enrollment management activity; (B: Right) The unified class model that results.

Example. Figure 6(A) depicts the UML activity and class diagrams of a CourseMan variant that manages enrollment, while Figure 6(B) illustrates the resulting unified class model of the activity. The unified class diagram comprises five domain classes and the realization of five activity-specific associations. The `EnrolmentMgmt` class functions as the activity class, while `DHelpOrSClass` serves as a decision class that captures domain-specific decision logic. The remaining three classes are data classes that correspond to three data stores, and they also correspond to three domain classes in the UML class diagram. The unified class model omits domain-specific associations.

The passage describes how the five associations in the unified class model from Figure 6(B) are used. Three of these associations link the `EnrolmentMgmt` activity class to the data classes, binding their modules to the containment tree of `ModuleEnrolmentMgmt`. The remaining two associations link the decision class `DHelpOrSClass` to two data classes, `SClassRegistration` and `HelpRequest`, which correspond to the data stores connected to the two action nodes branching from the decision node. These associations are considered weak dependency associations and are included in this case to allow the decision logic encapsulated by `DHelpOrSClass` to reference the two data classes.

4. Module Action Semantics

This section provides a formal definition of *module action* that is based on the UML Action language [5]. Our definition focuses on describing the structure of module action and its pre- and post-states. We recur-

sively define module action by beginning with the most primitive type of action called *atomic action*. We then combine these actions to form *atomic action sequence* and, more generally, *structured atomic action*, resulting in a precise specification of the behavioral semantics of modules in MOSA.

4.1. Atomic Action

Although each module is different, we observe that there exists a set of primitive behaviors that underlie all modules. We capture these primitive behaviors in what we term *atomic actions*.

Definition 3. *An **atomic action** is a smallest meaningful module behavior provided to an actor (which is either a human or another module/system) through the view for manipulating the domain objects of the domain class. Atomic action is characterised by:*

- `name`*: the action name.*
- `preStates` *(for* `localPrecondition` *[5]): the states at which a current module must be in order for this action to proceed.*
- `postStates` *(for* `localPostcondition` *[5]): the states at which the action completes its execution on a current module.*
- `fieldValSet` *(for* `input` *[5]): captures the input of the action. It is a set of pairs (f, v) where f is the name of a domain field of the domain class, and v is the value assigned to this field by the action.*
- `output`*: the domain class for object manipulation actions and empty for all other actions.*

  Although attribute `name` *uniquely identifies an action, for ease of exposition, we usually list two other attributes,* `postStates` *and* `fieldValSet`*, with* `name`*. Thus, we denote by $a = (o, s, i)$ an atomic action a whose* `name`*,* `postStates`*, and* `fieldValSet` *are o, s, and i (*resp.*). We use the dot notation to refer to the components, e.g., , a.*`postStates` $= s$*.* □

The above definition has the following points to note. Firstly, module states are used to abstract from the local pre- and post-conditions of each action, which allows for the flexible combination of actions based on states to construct more complex ones. A module state represents the states of the model, view, and controller components of a module as these components handle a module action. Certain module states can occur concurrently, which we refer to as concurrent states and are represented using the '+' operator. The `postStates` of a primitive action consist of a single state, while that of more complex actions (which will be discussed in Section 4.3) consists of multiple states.

Secondly, another important aspect of the actions is the input values they require. Since actions manipulate domain fields, their inputs need to correspond to the fields they modify. Thus, we define action inputs as a set of field-value pairs, which can be empty. The value in each pair can either be provided by the user or come from a previous action in a composed behavior. In the following subsections, we will discuss how actions can be combined to form more complex behavior.

Thirdly, the action output consists of, at most, one type, which corresponds to the domain class of the current module. It's worth noting that only the object manipulation actions have this output; other actions have an empty output because they do not produce any real output value.

Table 1 lists definitions of the core atomic actions. For exposition purposes, we divide the actions into two groups. The first group includes actions that concern the overall operational context of the module. The actions in this group include `open`, `newObject`, `setDataFieldValues`, `reset`, and `cancel`. The post-states of these actions consist of the following states: `Opened`, `NewObject`, `Editing`, `Reset`, and `Cancelled` (*resp.*). The second group includes three essential domain object manipulation actions: `createObject`, `updateObject`, and `deleteObject`. The post-states of these actions include the following states: `Created`, `Updated`, and `Deleted` (*resp.*).

Note from Table 1 that only action `setDataFieldValues` requires the `fieldValSet` to be specified as input. Other actions do not require any input and thus, for them, this set is empty. Note also how the two module states `ObjIsPresent` and `ObjIsNotPresent` can each occur concurrently with any one

Table 1: The core atomic actions

Name	Pre-states	Post-states	Description
`open`	{`Init`}	{`Opened`}	Open the module's view presenting the domain class.
`newObject`	{`Opened`, `Created`, `Updated`, `Reset`, `Cancelled`}	{`NewObject`}	Remove from the view any object currently presented and prepare the view for creating a new object.
`setDataFieldValues`	{`NewObject`, `Editing`, `Created`, `Updated`, `Reset`, `Cancelled`}	{`Editing`}	Set values for a sub-set of the view's data fields.
`createObject`	{`NewObject`, `Editing` + `ObjIsNotPresent`}	{`Created`}	Create a new object from data entered on the view. The created object is presented on the view.
`updateObject`	{`Editing` + `ObjIsPresent`}	{`Updated`}	Update the current object from data entered on the view. The updated object remains on the view.
`deleteObject`	{`Created`, `Updated`, `Reset` + `ObjIsPresent`, `Cancelled` + `ObjIsPresent`}	{`Deleted`}	Delete the current object. The deleted object is removed from the view.
`reset`	{`Editing`}	{`Reset`}	Initialise the view to redisplay the current object (discarding all user input).
`cancel`	{`NewObject`, `Editing` + `ObjIsNotPresent`}	{`Cancelled`}	Cancel creating a new object (discarding all user input, if any).

of the following states: `Editing`, `Reset`, and `Cancelled`. For example, the concurrent state `Editing` + `ObjIsPresent` means that the module is currently presenting an object on the view and that this object is being edited by the user. In contrast, `Editing` + `ObjIsNotPresent` means that the module is currently prompting the user to enter input data for a new object. This object has not yet been created.

4.2. Atomic Action Sequence (ASE)

In practice, the core atomic actions are combined in sequence to form more useful behavior. This behavior, which we call *atomic action sequence*, corresponds to an interaction scenario. We model this sequence using structured actions of the UML Activity diagram [5]. We denote by `first` and `last` two functions that return the first and last elements (*resp.*) in a sequence.

Definition 4. *An **atomic action sequence (ASE)** $S = (a_1, \ldots, a_n)$ is a module action iff* $a_i.\texttt{postStates} \subseteq a_{i+1}.\texttt{preStates}$ $(\forall a_i, a_{i+1} \in S)$. *$S$ has the following properties:*

- $S.\texttt{preStates} = \texttt{first}(S).\texttt{preStates}$
- $S.\texttt{postStates} = \texttt{last}(S).\texttt{postStates}$
- $S.\texttt{fieldValSet} = \texttt{first}(S).\texttt{fieldValSet}$
- $S.\texttt{output} = \texttt{last}(S).\texttt{output}$

Example. Figure 7 shows an ASE that creates a new domain object of type T in a module. This ASE consists of a sequence of four atomic actions and is characterized by: `name`="`Sequence : create objects`", `fieldValSet`=`setDataFieldValues.fieldValSet`=$\emptyset$, `postStates`={`Created`}.
The first atomic action is `open`, which opens the view presenting the domain class. Once completed, this action raises an event with the state `Opened`, allowing interested listeners of this event to handle it. This action then leads to the execution of the second atomic action:

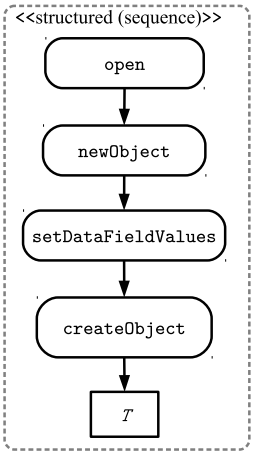


Figure 7: An ASE to create a domain object of type T.

`newObject`. This sequence is valid because, as listed in Table 1, `open.postStates` $\subset$ `newObject.preStates`. The `newObject` action prepares the view so that it is ready to receive input from the user for creating a new object. Once completed, this action raises an event with state `NewObject`. Because this state is contained in `setDataFieldValues.preStates`, we place `setDataFieldValues` as the third action of the ASE. This action is responsible for setting the values of all the view fields, which render the domain fields of the domain class. We place `createObject` as the next (and final) action of the ASE since `setDataFieldValues.postStates` $\subset$ `createObject.preStates`. This action creates a new domain object with the values of the view fields.

A useful property that arises from our concept of ASE is the natural multi-level nesting of ASE-backed behaviors along a path in the module containment tree. Specifically, an ASE S is considered 'nested' inside another ASE S' if there exists an activity edge that connects a member action of S' to the start action of S. In MOSA, S' is performed on the view of a composite module, and S is on the view of one of its child modules. For example, the ASE of `ModuleStudent` (shown in Figure 7) has a nested ASE that is performed on the child module of type `ModuleEnrolment`. The ASE of `ModuleStudent` itself is nested inside that of `ModuleSClass`, thus creating a 2-level nesting. The definition of ASE gives rise to the notion of *reachable state*, which is a module state that can be reached from a given action. We discuss this concept below and use it in the subsequent subsection to define a more generic action composition.

Definition 5. *A module state s' is **reachable** from an atomic action a if there exists at least one ASE whose first member action is a and whose post-state is s'. Action a is called the **source action** of s'.* □

Clearly, an atomic action can always reach its own post-state. Let us define the reachable states for the atomic actions shown in Table 1. First, the reachable states of the action `open` include `Opened`, `NewObject`, `Editing`, `Created`, `Updated`, `Deleted`, `Reset`, and `Cancelled`. This is because once the module's view is opened, it is ready to perform any of the core atomic actions (in some sequences). The rest of the core actions cannot reach the state `Opened` because this state is only raised once. Second, the reachable states of `newObject` include `NewObject`, `Editing`, `Created`, `Reset`, and `Cancelled`. Additionally, `newObject` cannot reach `Updated` and `Deleted` because this action is reserved for creating a new object and cannot lead to updating or deleting an existing object. Third, the reachable states of `setDataFieldValues` include `Editing`, `Created`, `Updated`, and `Reset`. This action cannot reach `NewObject`, `Deleted` and `Cancelled` because it concerns only input data and cannot initiate or cancel object creation, nor can it lead to object deletion. Finally, the remaining five actions each have only one reachable state, which is their own state. These actions are "stubs," in the sense that they terminate all the ASEs that lead to them.

Example. The ASE in Figure 7 shows that state `Created` is reachable from any of the three member actions that precede the action `createObject`. These include `open`, `newObject` and `setDataFieldValues`.

4.3. Structured Atomic Action (SAA)

More generally, we observe that a set of related ASEs form a *structured atomic action*. In essence, this action defines a generic behavior that consists of alternative interaction scenarios (each of which is specified by one ASE in the set) that are usually performed (possibly concurrently) by the user.

Definition 6. *A **structured atomic action (SAA)**,* w.r.t. *a source atomic action a and a set of post-states $E = \{s_1, \ldots, s_n\}$ reachable from a, is the set $A = \{S : ASE \mid \texttt{first}(S) = a, S.\texttt{postStates} \subseteq E\}$, where:*

- $A.\texttt{preStates} = a.\texttt{preStates}$
- $A.\texttt{postStates} = E$
- $A.\texttt{fieldValSet} = a.\texttt{fieldValSet}$
- $A.\texttt{output} = \bigcup_{S \in A}(S.\texttt{output})$

Abstractly, we write $A = (a, \{s_1, \ldots, s_n\}, i)$. If the `fieldValSet` i is $\emptyset$ then we omit it and simply write A as $(a, \{s_1, \ldots, s_n\})$. □

Clearly, SAA generalizes both atomic action and ASE: An ASE is a single-member SAA, while an atomic action a is the SAA $(a, \{a.\texttt{postState}\})$. Further, SAA is significantly shorter to compose than an ASE set: All we need to do is specify the start atomic action and the desired post-states.

Example. Let's consider the SAA $(\texttt{newObject}, \{\texttt{Created}, \texttt{Cancelled}\})$, which represents a common set of ASEs that start with the action `newObject` and end only when either the state `Created` or `Cancelled` is reached. The set consists of frequently-occurring ASEs, such as the one described earlier in Figure 7, but excluding the first action. We assume that the module's view is already opened. The remaining ASEs model alternative scenarios in which the user wants to cancel creating the object at some point between performing the `newObject` action and the `createObject` action.

5. Domain Behavior Patterns

This section explains our pattern-based approach to incorporating domain behaviors into a domain model. Each domain behavior, described at a high level using a UML Activity diagram and domain-model-based statements, is translated into a specification with two parts: (1) a part of the unified class model with new activity classes, and (2) the activity graph logic of the input activity and the mappings to connect the activity with the unified class model. We achieve this translation by applying domain behavior patterns, which are defined as corresponding to the five essential UML activity modeling patterns [4], serving as restrictions to sharpen AGL's semantics domain.

5.1. Specifying Domain Behavior Patterns

We are particularly interested in the design of the *pattern form* [25, 26]. To keep the patterns generic, we present for each pattern form a UML Activity diagram and a **template configured unified class model** that realizes it. The template model is a 'parameterized' configured DCSL unified model, in which elements of the non-annotation meta-concepts are named after the generic roles that they play. For brevity, we omit all associative fields and base domain methods from the model's diagram. Attached to the template model is an activity graph specification in the AGL (that is further explained in Section 6). The AGL specification aims to specify the activity graph logic of the input activity and to maintain the synchronization of current execution states (*w.r.t.* the activity) with current states of the domain (*w.r.t.* the unified class model).

We provide an example of each pattern in our catalog using a DCSL unified model for the CourseMan enrollment management activity. Due to space constraints, we will only illustrate the *Decisional* pattern in this paper. For more information on the other four patterns (*Sequential*, *Forked*, *Joined*, and *Merged*), please refer to the accompanying technical report [27].

The top-left of Figure 8 illustrates the UML Activity diagram, while the top-right shows the template configured unified model. Apart from the activity class `Ca`, this model includes five other domain classes, namely `Cd`, `D`, `C1`, `Cn`, and `Ck`, that are mapped to the five activity nodes. In particular, class `Ck` is a control class referenced by the control node c_k of the activity diagram. Class `D` is a decision class that implements the `Decision` interface. Since the decision's logic may require knowledge of the domain classes involved (namely `C1`, `Cn`, and `Ck`), there are optional weak dependency associations between `D` and these classes. Depending on the domain requirements, we may or may not need some of these associations.

The class `Ca` is associated with the other four domain classes in a one-to-many relationship. It is worth noting that the association with the class `Ck` can serve as a bridge to other activity flow blocks in a larger activity diagram. However, if c_k is a decision node, `Ck` has no associations, and the association to `Ck` is replaced or "unfolded" into a set of associations that directly connect `Ca` to the domain classes of the model containing `Ck`. In the template model, the two associations between `Cd` and `C1`, `Cn` indicate that both `C1` and `Cn` are aware of `Cd` due to the passing of object tokens from e_d to e_1 and e_n via the decision node.

The AGL specification for this pattern includes five `ANodes`. The first `ANode` creates a new `Cd` object, while the second `ANode` runs the decision logic. The third and fourth `ANodes` represent the two decision cases: The first creates a new `C1` object for the specified `Cd` object, and the second (which is repeated for all n) creates a new `Cn` object for the same `Cd`. The fifth `ANode` is used when `Ck` is specified and involves two variables, k and k_{out}, both of which are dependent on `Ck`. The variable k specifies the control node type, while the variable k_{out} specifies the array of output domain classes of `Ck`.

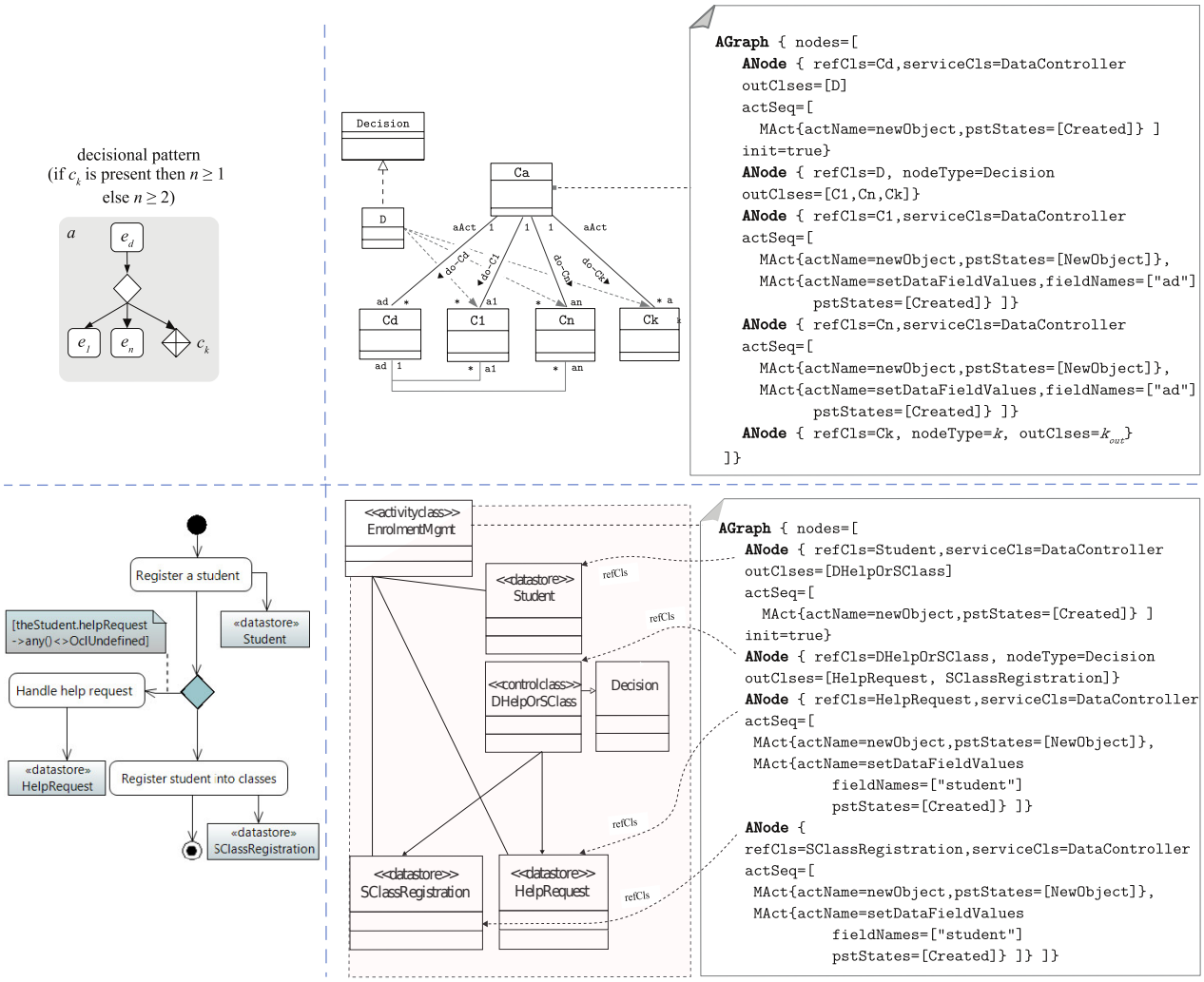


Figure 8: The decisional pattern form.

Example. The bottom of Figure 8 illustrates how this pattern is applied to CourseMan's enrolment management activity. In this example, we have: `Ca` = `EnrolmentMgmt`, `Cd` = `Student`, `D` = `DHelpOrSClass`, $n = 2$, `C1` = `HelpRequest`, `C2` = `SClassRegistration`. The control node c_k is not specified.

Figure 9 shows three snapshots of the GUI for this example. The first GUI is for student registration, while the second and third GUIs are for the cases where help request is and is not requested (*resp.*). The activity's GUI contains the GUIs of the three actions in separate tabs. In both cases of the decision, the `Student` object that is created in the first action (e.g., `Student`(`name`="Nguyen Van Anh")) is passed on to the next action. This object is then displayed in the data field of the associative field `student` of the domain class referenced by this action.

5.2. Applying Domain Behavior Patterns

To obtain a concrete AGL specification when applying a domain behavior pattern, basically, we follow three main steps:

1. Representing the input UML Activity diagram and the template configured unified model of the pattern as activity graphs;

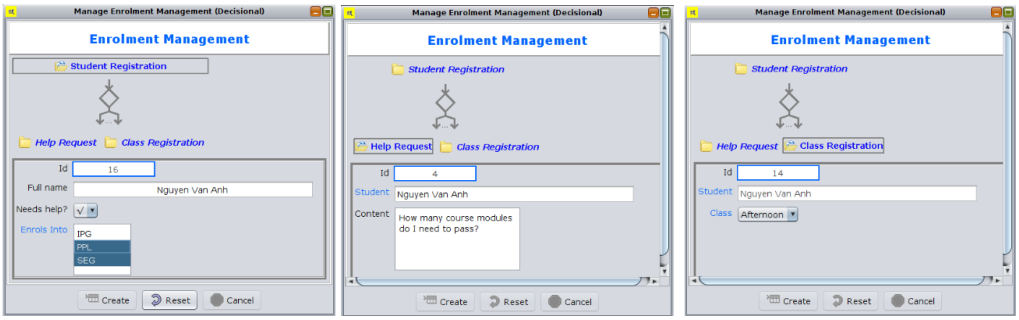


Figure 9: The decisional pattern form view of enrolment management.

2. Defining the matching of the template to the activity in order to define `ANodes`, their sequence, and the domain classes referenced by them. The reference from `ANodes` to domain classes (expressed with the keywords `refCls` and `outCls`) provides a mapping between the activity and the unified class model;

3. Expressing the generic behavior of the domain module *w.r.t.* the domain class referenced by each ANode through a SAA using the keyword `actSeq`. This ensures that the execution state of the activity is synchronized with the current state of the unified class model.

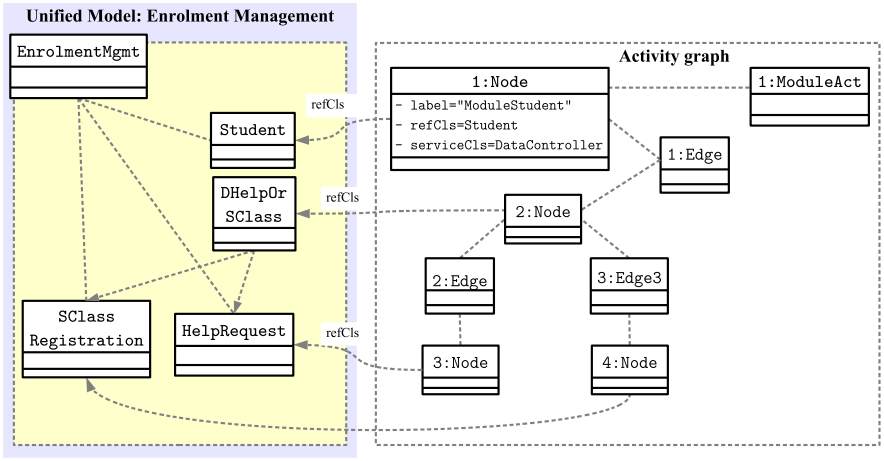


Figure 10: (LHS) A repeat of the unified class model shown in Figure 6; (RHS) The activity graph of this model.

Example. Figure 10 shows CourseMan's enrollment management activity graph on the right-hand side, while the corresponding unified class model is shown on the left-hand side. Tables 2(A) and (B) list the states of the nodes and edges (*resp.*) of the activity graph. Table 2(C) lists the `ModuleAct` objects that are referenced by the `Nodes` in Table 2(A). A `ModuleAct` object represents an SAA. Each table column lists the values of a representative field of an object. For instance, node 1:`Node` references the domain class `Student` (hence also references `ModuleStudent`) and has `serviceCls` = `DataController` (a default module service class developed as part of the jDomainApp framework [16]). The node also references object 1:`ModuleAct`. The `refCls` field of each node is depicted in the figure using a dashed curve labeled "`refCls`" that connects the node to the referenced domain class.

We provide Definition 7 to clarify how the software is generated in MOSA. This definition makes precise the general notion of module-based software that we introduced in Section 2.2 and takes into account the composition of a unified class model and an activity graph. It highlights the sub-set of modules that owns the activity classes and how these modules handle the execution of the associated activity graphs.

Table 2: (A: Top) `Node` objects, (B: Bottom-left) `Edge` objects of the activity graph in Figure 10 and (C: Bottom-right) `ModuleAct` objects that are referenced by the `Node`s

Node-Id	label	refCls	serviceCls	actSeq
1	"MStudent"	Student	DataController	[1:ModuleAct]
2	"MDHelpOrSClass"	DHelpOrSClass	null	null
3	"MHelpRequest"	HelpRequest	DataController	[2:ModuleAct, 3:ModuleAct]
4	"MSClassRegistration"	SClassRegistration	DataController	[4:ModuleAct, 5:ModuleAct]

Edge-Id	n1	n2
1	1:Node	2:Node
2	2:Node	3:Node
3	2:Node	4:Node

MAct-Id	actName	postStates	fieldNames
1	newObject	{Created}	
2	newObject	{NewObject}	
3	setDataFieldValues	{Created}	{"student"}
4	newObject	{NewObject}	
5	setDataFieldValues	{Created}	{"student"}

Definition 7. *Given a unified class model that contains a non-empty set of activity classes, each of which is attached to an AGL specification describing the activity graph logic of a domain activity. A unified domain model D is a composition of the unified class model and the AGL specifications. A software generated in MOSA* w.r.t. *D consists in a set of modules, each of which owns a domain class in D and the behavior of the* `newObject` *action of every owner module of an activity class includes the logic described by the activity graph that is configured by the AGL specification attached to that class.* □

6. Module-Based Domain Behavior Language

This section provides a brief overview of AGL as an aDSL for integrating domain behaviors into a domain model. The language allows the creation of activity graphs by configuring them directly on the domain model using annotations. To follow the meta-modeling approach for DSLs [6], an abstract syntax meta-model (ASM) and an annotation-based textual concrete syntax model (CSM) for AGL are defined. Only the syntactic aspects of AGL are explained here, as its behavioral semantics are characterized in Sections 4 and 5.

6.1. Abstract Syntax

We define the AGL's domain requirements by applying inclusion (I), exclusion (X), and restriction (R) clauses to the UML activity requirements as detailed in [5, p. 373]. Specifically, the following clauses apply:

- (I1) a module action, as discussed in Section 4, is a special form of action [5, p. 441];
- (R1) each executable node [5, p. 403] performs a sequence of module actions;
- (X1) using a variable with activity [5, p. 377];
- (X2) using variable actions [5, p. 469].

I1 and R1 are necessary to integrate the activity graph into MOSA. X1 and X2 exclude variable usage, which is an alternative to object flow, the primary means for moving data in UML activities [5, p. 377]. The AGL activity model captures current system state by directly referencing the unified class model, instead of using object flow.

Figure 11 depicts a simplified metamodel for AGL's abstract syntax. Specifically, the AGL metamodel includes meta-concepts adapted from the UML metamodel for the activity graph model: `ActivityGraph`, `Node`, `ControlNode`, and `Edge`. Several restrictions are defined on `Node` to form AGL's domain of activity graphs, which should be narrower (as a subset of) UML's domain of activity graphs.

The `Node` meta-concept represents activity nodes and has four properties. The first property is the attribute `label`, which represents the node label. The second property is `out`, which is derived from

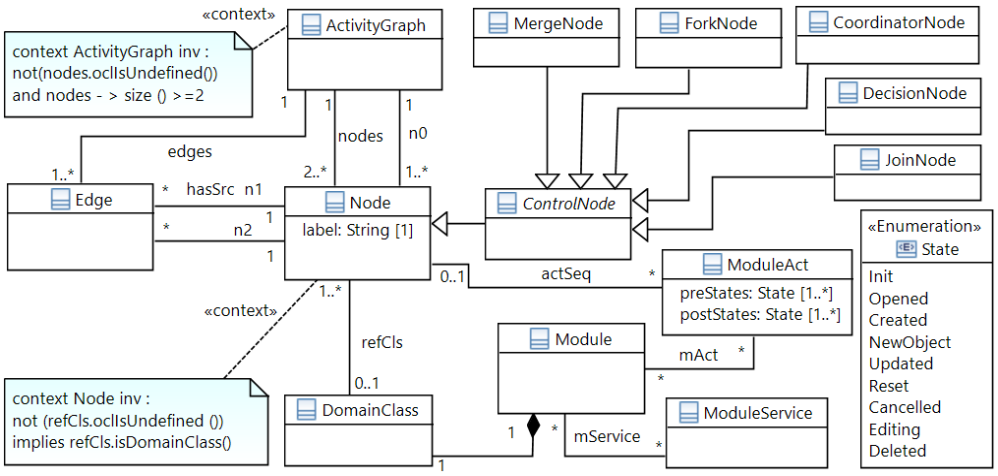


Figure 11: A simplified metamodel for AGL's abstract syntax.

the association `hasSrc(Edge, Node)` and records the outgoing edges from a `Node`. The next two properties specify the referenced module that is associated with this node. The `refCls` property references the domain class of the referenced module, while the `serviceCls` property specifies the actual `ModuleService` class of the module. The `serviceCls` property corresponds to the target role of the association from `Node` to `ModuleAct` and allows the current `Node` to perform the `ModuleAct`s specified by the property `actSeq`. The `CoordinatorNode` meta-concept is used to represent a task group and does not pass its data out to the outgoing edges. Instead, it passes the original input data out and the member tasks interact with each other to perform the group's logic. The coordinator's UI serves as the container of those of the member tasks, allowing the user to have a comprehensive view of the group. An example of this meta-concept is provided in Section 7.

Meta-concept `ModuleAct` represents SAA-typed module actions as defined in Definition 6. Note that we use an enumeration called `ActName` and an enumeration called `State` to represent the action names and the union of pre-states and post-states (*resp.*). `State`, in particular, represents both normal states and concurrent states (see Section 4.1).

We need to provide well-formedness rules (WFRs) for the metamodel to obtain valid AGL models. These rules can be divided into two parts: (1) WFRs from the UML metamodel for valid activity graphs; and (2) WFRs to ensure that AGL models are obtained as a composition of available domain behavior patterns. While Figure 11 shows two WFRs, a full specification of the WFTs in OCL for this metamodel can be found in the accompanying technical report [27].

6.2. Annotation-Based Textual Concrete Syntax

We aim to define an annotation-based textual concrete syntax for AGL. To achieve this, we define a metamodel for the concrete syntax (CSM) of AGL by transforming the abstract syntax meta-model (ASM). Figure 12 illustrates the annotation-based CSM, which is suitable for embedding into a host OOPL. Moreover, we strive for a concise CSM that utilizes a small set of annotations. From a practical perspective, such a model is desirable as it results in a compact concrete syntax that requires less effort from the language used to construct a unified domain model.

To achieve the transformation from ASM to a meta-model suitable for annotation-based representation, we follow two steps. In the first step, we transform ASM into the meta-model CSM_T, which is compact and appropriate for annotation-based representation. CSM_T comprises three meta-concepts, namely activity graph (`AGraph`), activity node (`ANode`), and module action (`MAct`). In the second step, we transform CSM_T into the actual annotation-based CSM, which is "embedded" into the host OOPL. This CSM is constructed using three OOPL meta-concepts, namely `class`, `annotation`, and `property`. Since the CSM is embedded directly into the OOPL, its structure defines the core structure of a CSM model of the AGL's textual syntax. We argue that the CSM of AGL will contain additional meta-concepts that help describe the structure of

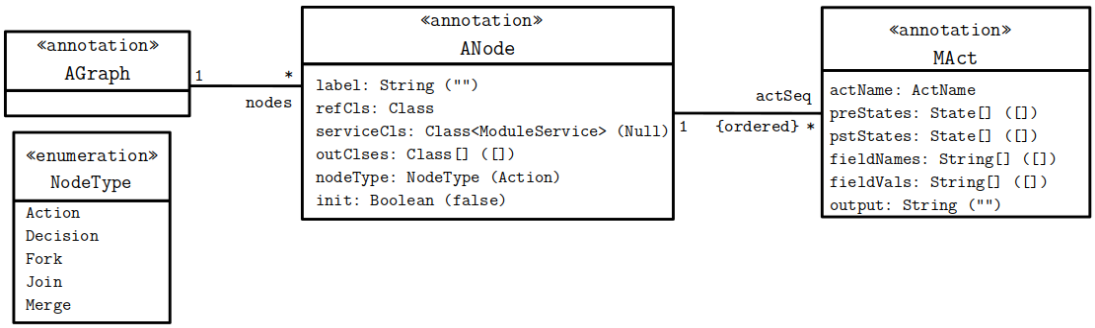


Figure 12: Illustration for AGL's annotation-based textual concrete syntax, realized in Java.

the BNF grammar rules, following the concrete syntax meta-modeling approach [6]. The textual syntaxes of Java and C# are both described using this grammar. For a detailed explanation of the transformation ASM $\rightarrow$ CSM$_T$, we refer the reader to Appendix B of the technical report [27] accompanying this paper.
Example. The bottom-right part of Figure 8 depicts the unified class model of CourseMan's enrollment management activity. The AGL specification for this activity is defined by an `AGraph` element, which is written inside a note box that is attached to the `EnrolmentMgmt` activity class in the unified class model. For the purpose of this paper, we will write the AGL specification in text using the structured note box notation of DCSL, as described in Section 2.3.

7. Case Study and Tool Support

This section introduces a complex case study, the order management domain (OrderMan), which is adapted from the OMG/UML specification [5, p. 396]. The purpose is to demonstrate the practical application of AGL and investigate how our proposed method can be used to develop software for real-world problem domains. Additionally, this section presents the tool support used, focusing on explaining the primary design choices and technologies utilized.

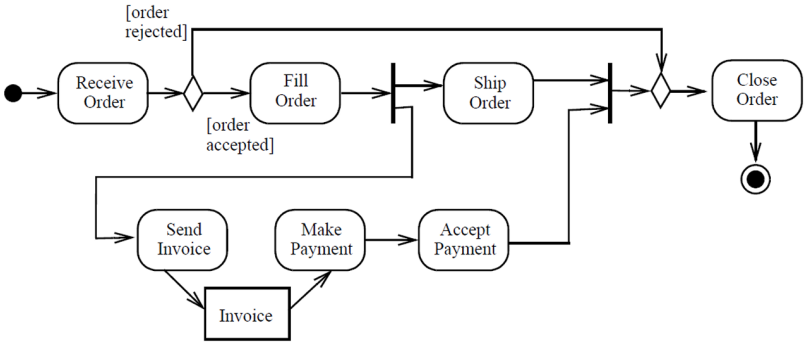
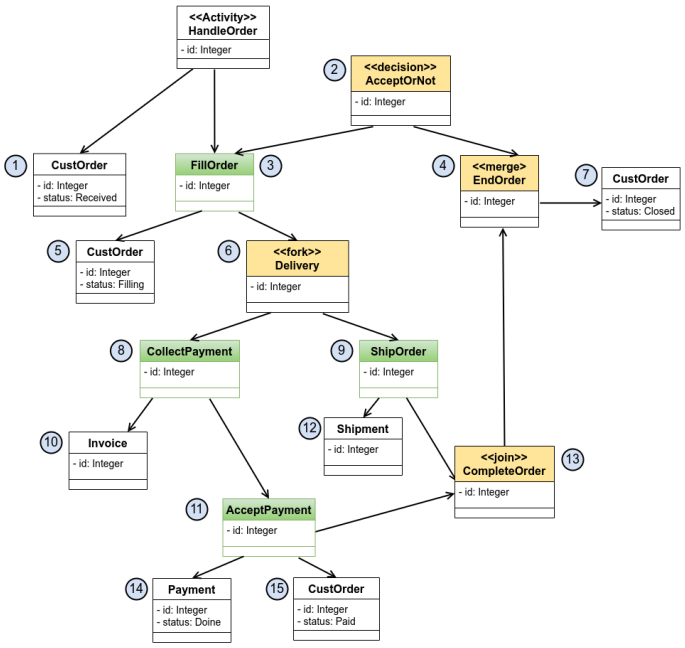


Figure 13: The UML activity diagram for the process to handle orders, adapted from [5, p. 369].

7.1. Case Study: OrderMan

Figure 13 illustrates the UML Activity diagram that depicts the process of handling orders in OrderMan. To generate the unified domain model for OrderMan, which is shown in Figure 14, we applied all the domain behavior patterns from our first catalog. Figure 14(A) presents an activity graph with nodes labeled with numbers that correspond to the activity HandleOrder. It also shows part of the unified class model, consisting of one activity class (HandleOrder), four primary data classes (`CustOrder`, `Invoice`, `Shipment`, `Payment`), four control classes (`AcceptOrNot`, `Delivery`, `CompleteOrder`, `EndOrder`), and four remaining

classes related to coordinator nodes (`FillOrder`, `CollectPayment`, `ShipOrder`, `AcceptPayment`). As discussed in Section 6, coordinator nodes are used to provide a complete view of a task group. For instance, the coordinator node FillOrder (node 3) manages two tasks, namely UpdateOrder (node 5) and DeliveryOrder (node 6). The FillOrder node simply coordinates the tasks to ensure that UpdateOrder is performed before DeliveryOrder. It does not contribute any data to the flow, but it enables the user to observe and execute the task flow on the UI.



1:AG (HandleOrder)

Node	label	nodeType	refCls	serviceCls	outNodes	actSeq
1	MOrder	-	CustOrder	default	[2:Node]	[1:MAct]
2	MAcceptOrNot	Decision	AcceptOrNot	-	[3:Node, 4:Node]	-
3	MFillOrder	Coordinator	FillOrder	default	[5:]	[2:, 3:, 4:]
4	MEndOrder	Merge	EndOrder	-	[7:]	-
5	MOrder	-	CustOrder	default	[6:]	[5:]
6	MDelivery	Fork	Delivery	default	[8:, 9:]	[2:, 4:]
7	MOrder	-	CustOrder	default		[6:, 7:]
8	MCollectPayment	Coordinator	CollectPayment	default	[10:]	[2:, 3:, 4:]
9	MShipOrder	Coordinator	ShipOrder	default	[12:]	[2:, 3:]
10	MInvoice	-	Invoice	default	[11:]	[2:, 3:]
11	MAccept Payment	Coordinator	AcceptPayment	default	[14:]	[2:, 4:, 8:]
12	MShipment	-	Shipment	default	[13:]	[2:, 3:]
13	MComplete Order	Join	CompleteOrder	-	[4:]	-
14	MPayment	-	Payment	default	[15:]	[2:, 8:]
15	MOrder	-	CustOrder	default	[13:]	[9:, 5:]

MAct

Id	actName	postStates	filterType	fieldNames
1	newObject	{Created}		
2	newObject	{NewObject}		
3	setDataFieldValues	{Created}		{"order"}
4	createObject	{Created}		
5	showObject	{Updated}		
6	showObject			
7	updateObject	{Updated}		
8	setDataFieldValues	{Created}		{"invoice"}
9	filterInput		FilterCustOrder FromPayment	

Figure 14: (A: Left) The activity graph whose nodes are labeled with activity and component classes; (B: Top-right) The `Node` objects; (C: Bottom-right) `ModuleAct` objects that are referenced by the `Nodes`.

Figure 14(B) showcases how each node of the activity graph is mapped to a corresponding module (*w.r.t.* the domain class referenced by `refCls`), the out nodes (*w.r.t.* `refCls`), and the `ModuleAct` objects that specify SAAs for the module's behavior. These `ModuleAct` objects, along with SAAs, are listed in Figure 14(C). To develop the OrderMan software with the GUI, as demonstrated in Figure 15, the unified class model that integrates the AGL specification for the activity model must be encoded in Java. The implementation for OrderMan can be found in the git repository[1]. A detailed explanation of this implementation is presented in Section 7.2.

7.2. Tool Support

The implementation of our method is demonstrated in Figure 16, using a supporting tool built on jDomainApp, a Java software framework we have previously discussed in [4]. The tool can be accessed at the git repository[2]. To generate software from the input unified domain model, as described in Section 3

[1] https://github.com/jdomainapp/orderman
[2] https://github.com/jdomainapp/jda-mbsl

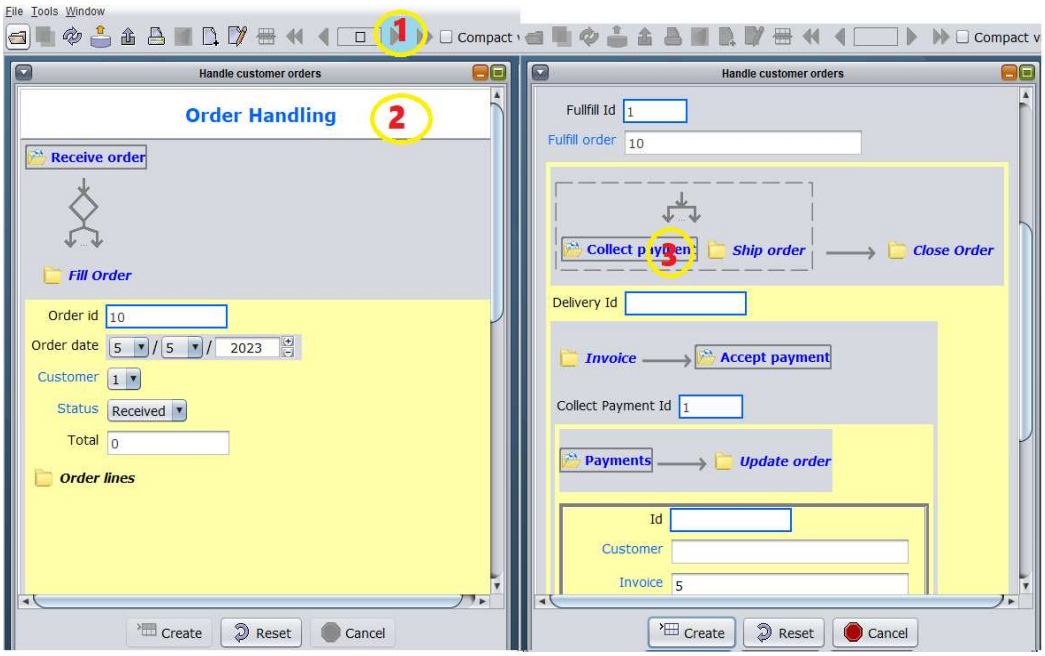


Figure 15: The GUI of the OrderMan software generated by the tool.

and Definition 7, the developer must encode the unified domain model in DCSL and AGL, resulting in a Java program that has two main components: (1) The DCSL unified model, which includes component and activity classes, as explained in Definition 2; (2) The AGL specification for the activity graphs associated with the activity classes. It is important to note that both the DCSL and AGL specifications are encoded in Java, as depicted in the middle and right panels (*resp.*) of Figure 16.

The tool consists of three main components: the model manager, view manager, and object manager. The model manager registers and makes the unified class model available to other components. The left window of Figure 16 displays a list of module classes and their corresponding domain classes that are attached with an annotation `@AGraph` for AGL specification, as presented in the middle window. The view manager automatically generates the GUI of the software and handles user interaction based on the configuration descriptions of modules specified in MCCL [4]. For instance, the module class `ModuleHandleOrder` that owns `HandleOrder` is specified with a module description in MCCL, as shown in the right window. Finally, the object manager manages the run-time object pool of each domain class and provides a generic object storage component that supports file-based and relational database storage. The relational data model is automatically generated from the unified class model when the software runs for the first time.

Listing 1: The activity class `HandleOrder` in Java

```
/**Activity graph configuration in AGL */
@AGraph(nodes={...
  /* 14 */
  @ANode(label="14:Payment", zone="11:AcceptPayment",
    refCls=Payment.class, serviceCls=DataController.class,
    outNodes={"15:CustOrder"},
    actSeq={
      @MAct(actName=newObject, endStates={NewObject}),
      @MAct(actName=setDataFieldValues, attribNames={"invoice"},
        endStates={Created})
    }), ...
})
/**END: activity graph configuration */
public class HandleOrder {...}
```

Listing 2: Java implementation of the annotation `AGraph`

```
package jda.modules.mbsl.model.graph.meta;

import java.lang.annotation.*;
import jda.modules.mbsl.model.graph.ActivityGraph;

@Retention(RetentionPolicy.RUNTIME)
@Target(value=java.lang.annotation.ElementType.TYPE)
@Documented
public @interface AGraph {
  ANode[] nodes();
}
```

We will further clarify how a module class (*w.r.t.* an activity class) can manage the execution of the activity graph (*w.r.t.* the activity class) based on its AGL specification. Let's consider a specific scenario with the `ModuleHandleOrder` module and the `HandleOrder` activity class, implemented as shown in Listing 1.

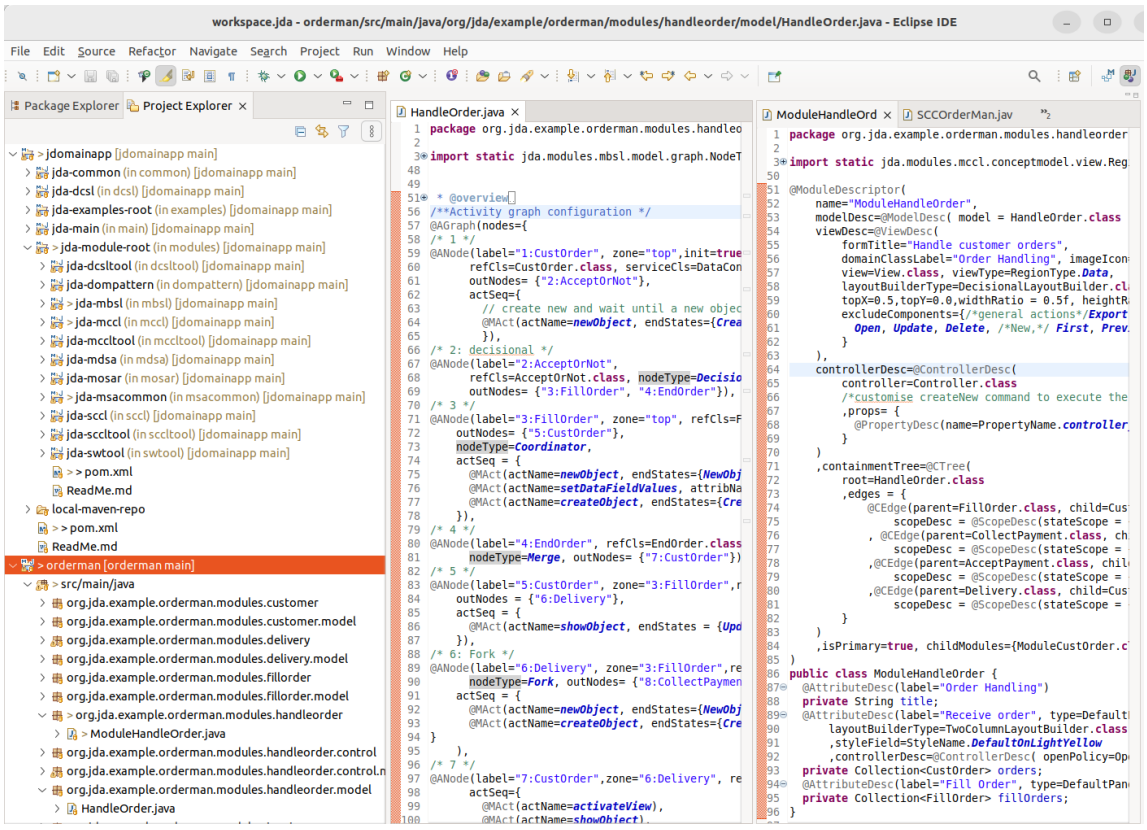


Figure 16: Illustration for the JDomainApp-based realization and usability of AGL.

When the software runs, an instance of the `ModuleHandleOrder` module is called. Using the module's configuration description presented in the right window of Figure 16, an `ActivityModel` is created as a composition of the `HandleOrder` object *w.r.t.* the `ModuleHandleOrder` module and the activity graph specified by the AGL for the `HandleOrder` activity class. Using the annotation mechanism in Java, the `HandleOrder` object can be treated as an `AGraph` object, allowing it to represent and manage the activity graph. The `AGraph` object enables the `ModuleHandleOrder` module to handle each of its `ANodes`, such as the `ANode` *w.r.t.* node 14 in Listing 1, as well as the domain class referred to by the `ANode`, in this case, the `Payment` class. For a more detailed implementation of this activity model, please refer to Appendix C in the accompanying technical report [27].

We will further explain how the AGL and its annotations is realized in Java. The AGL specification of an activity class is represented as an annotation of the activity class in Java. For example, Listing 1 illustrates the annotation used to express the AGL specification of the `HandleOrder` activity class. All the annotations in AGL, as summarized in Figure 12, have to be defined in Java. Listing 2 provides an example of such a definition for the `AGraph` annotation, and the other annotations in AGL are defined similarly.

8. Evaluation

In this section, we discuss an evaluation of AGL. We aim to point out that AGL is both essentially expressive and practically usable. We follow the guidelines defined in [28] in order to enhance the rigor of this experiment. The section closes a discussion with several remarks concerning the evaluation.

8.1. Research Questions

We evaluate the AGL+ language, which incorporates AGL into DCSL. We use the criteria defined in [29] to evaluate AGL+ as a domain-specific language. These criteria are *expressiveness*, *required coding level*, and *constructibility*. They measure the extent to which a language can express the properties of interest of its domain, allows expressing the underlying properties of the domain without too much hard coding, and provides facilities for building complex specifications in a piecewise, incremental way (*resp.*). To achieve this goal, we pose the following research questions.

RQ1. How effective is our approach in representing domain models in comparison to existing DDD methods?

RQ2. How much effort is required to define unified domain model in AGL+ for generating DDD software?

8.2. Case Study Design

Setup. We have selected three software applications (apps) that are derived from real-world requirements for our analysis. These applications are CourseMan, ProcessMan, and OrderMan. Although we have already elaborated on the third app in Section 7, we will provide further explanations on the other two apps.

The first application, CourseMan, is a software designed for managing courses in a university. The problem domain related to this case is relatively complex, representing a real-world problem domain that is well-suited for applying DDD. Additionally, this domain is a common and generic problem domain in academic institutions that has been studied in the literature [4, 30, 31].

The second app, named ProcessMan, manages the processes of the Faculty of IT at Hanoi University for teaching subjects (a.k.a. course modules) to students every semester and formally assessing their performances. This app was previously studied and developed using DCSL, as reported in our earlier work [8]. For this experiment, we have developed the current version of this app using the AGL-based method.

Measure. To evaluate the expressiveness of our method, we view it as an implementation of DDD patterns such as *Entity*, *Aggregate*, *Value Object* and *Service* (as described in [1]). Therefore, we measure how well our language can be mapped to these patterns. Additionally, we compare our approach to existing DDD methods, including two annotation-based extensions of DDD frameworks. The two frameworks that support commonly used third-party annotation sets are labeled as AL for ApacheIsis[3] [32] and XL for OpenXava [33].

We evaluate AGL+ 's expressiveness in two aspects: structural modeling and behavioral modeling. In terms of structural modeling, we use the four criteria of *domain class*, *domain field*, *associative field*, and *domain method*, which we previously used in our work on DCSL [4]. In terms of behavioral modeling, we compare our language to UML Activity diagram (labeled as AD), which is a widely-used method for capturing domain behaviors.

For measuring the required coding level (RCL), we also evaluate it in two aspects: structural modeling and behavioral modeling. In terms of structural modeling, we follow the approach reported in our previous work [4], using two sub-criteria: maximum number of lines of code (max-locs) and typical number of lines of code (typical-locs). The lower the values of these criteria, the better. Since DCSL, AL, and XL are all based on a host OOPL, we consider an annotation property as a line of code. Therefore, max-locs (typical-locs) represent the maximum (typical) number of properties needed to express a typical domain term. For example, to specify a typical domain field such as `Student.name` in DCSL, we need a maximum of three DAttr's properties (`length`, `min`, and `max`) or typically one of these properties.

To measure RCL from the behavioral perspective, we cannot compare AGL's RCL with that of other languages as AGL is the first aDSL of its kind. Hence, we assess AGL's RCL based on the compactness of the language's CSM, as explained in Section 6. This is determined by the reduction in the number of features in the CSM via the transformation ASM $\rightarrow$ CSM$_T$. Specifically, AGL's RCL is the percentage of the number of CSM$_T$'s features over the number of ASM's. A smaller percentage indicates a higher

[3] with the new name 'causeway' since March 26 2023

reduction in the number of features in the CSM and, consequently, a more compact CSM. Additionally, we consider quantitative measures of the number of domain patterns (modules) employed (defined) for building the underlying apps. The more patterns are employed, the less effort is required to obtain the software.

8.3. Results

This section explains the results of our experiment, answering the raised research questions.

RQ1. How effective is our approach in representing domain models in comparison to existing DDD methods?

As presented in Table 3(A), our language and the existing DDD approaches (AL and XL) partially implement the DDD patterns [1]. It should be noted that in DDD, the concept of a service is referred to as a building block and is defined as follows: "When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as a standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless" [1]. The first three patterns in our method are implemented using the aDSL DCSL, while the other approaches implement them using the UML Class diagram enriched by OCL. Although the UML Activity diagram can be utilized to implement the final DDD pattern, it requires additional mechanisms such as fUML [34] and OCL [7] to obtain an integrated semantic model of the behavioral and structural diagrams (one of which represents the domain model). Within our approach, services are represented as activity models, each consisting of an activity domain class (expressed in DCSL) and an activity graph (specified in AGL).

Table 3: (A-left) Criteria 1: The expressiveness criteria based on DDD patterns; (B-right) Criteria 2: The expressiveness criteria based on domain meta-concepts

Criteria 1	AGL+	AL	XL	AD	Criteria 2	AGL+	AL	XL	AD
Entity & Aggregate	DomainClass (DC)	√	√	✗	DC	**1/1**	1/1	0/1	✗
	DomainField (DF)	√	√	✗	DF	**8/8**	4/8	5/8	✗
	AssociativeField (AF)	√	√	✗	AF	**7/7**	0/7	1/7	✗
	DomainMethod (DM)	√	√	✗	DM	√	✗	✗	✗
Value Object	ImmutableDomainClass	√	√	✗	ADC	√	✗	✗	√
Service	ActivityDomainClass (ADC)	✗	✗	√					

Table 3(B) presents the evaluation results for AGL+ and existing DDD approaches (AL, XL, and AD). Each fraction in the table represents the ratio of the number of essential properties of the meta-attribute involved in the underlying meta-concept that are supported by AL or XL. The denominator of the ratio is the total number of essential properties. In our previous work [4], we defined the essential properties *w.r.t.* the meta-attribute `DAttr`, which includes eight properties: `unique`, `optional`, `mutable`, `id`, `auto`, `length`, `min`, and `max`. Therefore, a ratio of 4/8 for AL *w.r.t.* the term `DomainField` means that AL only supports four out of the eight properties of the meta-attribute `DAttr` that are used in `DomainField`. These four properties are `Column.allowsNull`, `Property.editing`, `PrimaryKey.value`, and `Column.length`. Similarly, there are seven essential properties *w.r.t.* the meta-attribute `DAssoc`: `ascName`, `ascType`, `role`, `endType`, `associate.type`, `associate.cardMin`, and `associate.cardMax`. It's worth to note that the ratio aims to evaluate the current approaches from a structural aspect, which coincides with the evaluation step for DCSL reported in our previous work [4]. For a more detailed explanation, please refer to Appendix D of the accompanying technical report [27].

As shown in Table 3(B), AGL+ is more expressive than AL and XL in both structural and behavioral aspects. These two languages only partially support structural aspect and they do not support behavioral aspect. In particular, AL and XL's support for AssociativeField is very limited compared to AGL+. AD's expressiveness of domain behaviors is better than AGL, but again the behavioral model lacks an explicit connection to structural models.

RQ2. How much effort is required to define unified domain model in AGL+ for generating DDD software?

It is evident from Figures 11 and 12 that AGL's RCL is $\frac{3}{10} = 30\%$, as defined in Section 8.2. Figure 11 shows that there are 10 meta-concepts of ASM involved in the transformation, excluding the four

meta-concepts (`DomainClass`, `Module`, `ModuleService`, and `State`) that are directly transferred to CSM_T. Figure 12, on the other hand, shows that the transformation results in only three meta-concepts, including `AGraph`, `ANode`, and `MAct`. Therefore, AGL enables the creation of a CSM that reduces the number of required meta-concepts for writing an AGL specification to only one-third.

Table 4 shows the number of domain behavior patterns (modules) employed (defined) for obtaining the apps CourseMan, ProcessMan, and OrderMan. While the number of modules indicates the complexity of the software, the number of domain behavior patterns highlights the applicability of our first catalog of patterns as well as the reduced effort required by developers to manually code the software.

Table 4: Statistics of Behavior Patterns and Modules for CourseMan, ProcessMan, and OrderMan

	Patterns	Modules
CourseMan	3	6
ProcessMan	5	37
OrderMan	5	13

8.4. Discussion

Let us conclude our evaluation with a number of remarks on the AGL-based method as follows.

Integrating the* AGL+*-based method into the software development process is crucial for its practical implementation. The method is suitable for integration into iterative [35] and agile [36] development processes. In this process, the development team, consisting of domain experts and developers, would utilize the AGL+ tool to work collaboratively on developing the unified domain model incrementally and generating software from this model. The domain experts provide feedback on the model via the software GUI, and the update cycle continues. The software prototypes generated can be used as intermediate releases for the final software. Additionally, in both processes, *model-driven software engineering (MDSE)* tools and techniques can be applied to increase productivity and address platform variability. Specifically, PIM-to-PSM model transformation [37, 38] would be utilized to automatically generate the unified domain model from a high-level one constructed using a combination of UML Class and Activity diagrams.

The usability of the software GUI is essential from the domain expert's perspective and impacts the usability of our method. Although we did not discuss this issue in this paper, we believe that two aspects of the software GUI - simplicity and consistency - contribute to its learnability [39]. We plan to evaluate GUI usability thoroughly in future work. The GUI design is simple because it directly reflects the domain class structure, as discussed in [4]. This representation is the most fundamental of the domain model. In addition, the GUI is consistent in presenting the module view and handling user actions performed on it. Consistency in presentation is achieved by applying a reflective layout to all module views, while consistency in handling is achieved by providing a common set of module actions (see Section 4) on the module view.

Constructibility. For AGL, the language's embedment in the host OOPL allows it to take for granted the general construction capabilities of the host language platform and those provided by modern IDEs (e.g., Eclipse). More specifically, using an IDE a developer can syntactically and statically check an AGL specification at compile time. In addition, she can easily import and reference a domain class in the AGL specification and have this specification automatically updated (through refactoring) when the domain class is renamed or relocated. Further, we would develop automated techniques to ease the construction of the AGL specification. Intuitively, for example, a technique would be to generate a default AGL specification for an activity and to allow the developer to customize it. We plan to investigate techniques such as this as part of future work.

Generality. The core of our mechanism for incorporating behaviors into a unified domain model is to use domain behavior patterns in MOSA. Since domain behavior patterns and MOSA do not depend on the implementation language, our method could be implemented using another object-oriented programming language as an alternative to Java. The language chosen should support the annotation mechanism, like Java, so that AGL+ could be implemented on top of it. Alternatively, instead of using annotation-based DSLs, we could define external DSLs to implement our method. These DSLs could be implemented using current language frameworks such as Xtext [40], GEMOC [41], and Monticore [42].

9. Threats to Validity

This section aims to identify potential threats to the validity of both our proposed method and evaluation method. We have categorized the threats into three groups of validity, as proposed by [43]: construct validity, internal validity, and external validity.

9.1. Evaluation Method

The composition for a unified domain model in terms of the unified class model and an activity graph model (see Section 6) follows a language composition approach described by Kleppe [6]. In this approach, the composition is formed by language referencing. That is, one component language (called *active language*) references the elements of the other component language (called the *passive language*). In our method, AGL is the active language and DCSL is the passive one.

The evolution of languages (including both AGL and DCSL) is inevitable if we are to support more expressive domain modeling requirements. We discuss in [4] how DCSL is currently expressive only *w.r.t.* an essential set of domain requirements that are found to commonly shape the domain class design. We argue that DCSL would evolve to support other structural features. For AGL, its ASM would be extended to support other activity modeling features, such as activity group [5, p. 405].

The selection of the unified modeling patterns used in our expressiveness evaluation is based on UML Class and Activity diagrams that we currently use to construct the unified domain model. A question then arises as to the adaptability of our method to other UML behavioral diagrams (e.g., State Machine and Sequence diagram). We plan to investigate this as part of future work.

9.2. Construct validity

In our case study, we have assumed that there are no misinterpretations of the domain requirements that would lead to an unsatisfactory model. In practice, the designer and domain expert would need to work closely with each other to ensure that the models are satisfactory. Our method helped mitigate the threat of misinterpretation by allowing the combination of a domain class model with a behavioral model (e.g., a UML Activity diagram), which constructed a unified domain model within a domain-driven architecture.

9.3. Internal validity

Internal validity threats refer to factors that may affect the accuracy and reliability of our results. One potential threat is the translation process from a high-level domain specification, expressed using UML Class and Activity diagrams, to an AGL+ specification. To address this, we provided a guideline in Section 5 for applying domain behavior patterns. Another possible threat is that the available domain behavior patterns may not cover the input UML Activity diagrams. To mitigate this, we evaluated our method on three complex apps (CourseMan, ProcessMan, and OrderMan) and plan to supplement our pattern catalog with additional patterns that can be applied to other types of complex apps. Additionally, we acknowledge that errors may occur during the implementation of AGL+ with the Java framework jDomainApp. To address this, we thoroughly tested and reviewed the code, and have made it available online for the community to use and report any issues.

9.4. External validity

Threats to the external validity of our method include those that affect its applicability to the development of other MSA-based [44] and DDD-based software with similar characteristics. The first threat stems from the fact that our method is designed for systems based on MDSA (a combination of MSA and DDD). The second threat concerns the generality of our case study. One could argue whether the case study we selected is representative of real-world scenarios. However, our pattern-based approach helps mitigate this threat because it is based on the well-known software design principle of keeping patterns generic.

10. Related Work

We position our work at the intersection between the following areas: DSL engineering, DDD, model-driven software engineering (MDSE), and attribute-oriented programming (AtOP).

DSL Engineering. DSLs [45, 46] can be classified based on the domain [6], as vertical or horizontal, or based on the relationship with a host language [18, 45, 46], as internal or external. Our proposed AGL is a type of fragmentary, internal, and horizontal DSL. The shared features that are captured in AGL are those that form the activity graph domain. As far as we know, AGL is the first aDSL defined for this purpose.

DDD. The idea of combining DDD and DSL to raise the level of abstraction of the target code model has been advocated in [18] by both the DDD's author and others. However, the work in [18] does not discuss any specific solutions. In this paper, we extended the DDD method [1] to construct a unified domain model. We combine this with an activity graph model to operate in a module-based software architecture. The unified class model and the activity graph model are expressed in two aDSLs (DCSL and AGL, *resp.*).

Behavioral modeling with UML Activity diagram. Although in his book [1] Evans does not explicitly mention behavioral modeling as an element of the DDD method, he does consider object behavior as an essential part of the domain model and that UML Iteraction diagrams would be used to model this behavior. Although in the book, Evans only uses Sequence diagrams as an example, in the ApacheIsis framework [2, 32] that directly implements the DDD's philosophy, a simple action language is used to model the object behavior. This language is arguably a specific implementation of the action sub-language [5, p. 441] of UML Activity diagram. It leverages the annotation construct of OOPL to specify a class operation with a pre-defined behavioral type. However, ApacheIsIs lacks support for a behavioral modeling method. Our combination of two aDSLs in this paper helps fill this gap. Our definition of module action in this paper incorporate the notion of state, which is more formally modeled in another UML behavioral modeling language called Behavior State Machines (BSM) [5, p. 305]. As discussed in Section 4, our notion of module action's pre- and post-states looks at a similar view with BSM. The difference is that our notation emphasizes the actual behavior, while BSM focuses on the behavior's effects in terms of states and state transitions.

Unified modeling with UML diagrams. Recent research has attempted to combine UML structural and behavioral diagrams to construct a system model, similar in spirit to the unified domain model that we proposed in this paper. The work in [47, 48] discusses combining UML class and state machine diagrams to model the system. Another work [49] explains the relationships between UML structural and behavioural diagrams and how these relationships can be leveraged to build a complete system model. Our proposed unified domain modeling is novel in that it combines UML Class and Activity diagrams by incorporating the domain-specific structure (activity classses) into the Class diagram for a unified class model. The unified class model and activity graph are connected by virtue of the fact that nodes in the graph execute actions of the modules that own the domain classes in the model. Our method is novel in the treatment of MVC. We basically use it at the 'micro' level to design each software module as a self-contained MVC component. We then expose a module interface and combine it with the activity graph design.

MDSE. The idea of combining MDSE with DSLs is formulated in [6, 38]. This involves applying the meta-modeling process to create meta-models of software modeling languages (include both general-purpose languages and DSLs). Our AGL's specification follows the pattern-based meta-modeling approach, but targets internal DSL. Our method is similar to the method proposed in [50, 51] in the use of a combination of DSLs to build a complete software model. However, our method differs in two technical aspects. First, we use (internal) aDSLs as opposed to external DSLs. Second, our method (being a DDD type) clearly highlights the boundary of the domain model and, based on this, proposes to use only two aDSLs. The above work uses four DSLs and does not clearly indicate which ones are used for constructing the domain model and which are used to build other parts of the software model.

AtOP. With regards to the use of AtOP in MDSE, a classic model of this combination is used in the development of a model-driven development framework, called mTurnpike [21]. More recently, the work in [24] proposes a bottom-up MDSE approach, which entails a formalism and a general method for defining annotation-based embedded models. Our method differs from both [21, 24] in two important ways: (1) the combination of two aDSLs that can be used to express the unified domain model, and (2) how this model is used to automatically generate the entire software.

11. Conclusion

In this paper, we introduce a unified modeling method for developing object-oriented, domain-driven software. Our approach involves constructing a unified domain model within the MOSA architecture by: (1) extending the conventional domain model, which is expressed in our previously developed aDSL, DCSL, to include domain-specific features from the UML Activity diagram; and (2) creating an additional aDSL, AGL, to express domain behaviors for the unified domain model and attaching its activity graph to the activity class using the annotation attachment feature of the host OOPL. We systematically developed a compact, annotation-based syntax for AGL using UML/OCL and a transformation from the conceptual model of the activity graph domain. We implemented our method as part of a Java framework and demonstrated the effectiveness of AGL+ in designing real-world software.

Our approach significantly advances the state-of-the-art in DDD by bridging the gap between model and code and constructing a unified domain model. Our proposed aDSLs are horizontal DSLs, which can be used across different real-world software domains. In the future, we plan to develop an Eclipse plug-in for our method and create graphical visual syntaxes for AGL+. We also intend to develop a technique for automatically transforming high-level behavioral models into AGL+ specifications.

Acknowledgments

This work is funded by the Vietnam Ministry of Education and Training under grant number B2022-NHF-01. We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.
[2] Dan Haywood, Apache Isis - Developing Domain-driven Java Apps, Methods & Tools: Practical knowledge source for software development professionals 21 (2) (2013) 40–59.
[3] J. Paniza, Learn OpenXava by Example, CreateSpace, Paramount, CA, 2011.
[4] D. M. Le, D.-H. Dang, V.-H. Nguyen, On Domain Driven Design Using Annotation-Based Domain Specific Language, Computer Languages, Systems & Structures 54 (2018) 199–235. doi:10.1016/j.cl.2018.05.001.
[5] OMG, Unified Modeling Language version 2.5.1 (2017).
[6] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2008.
[7] OMG, Object Constraint Language Version 2.4 (2014).
[8] 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 (2020) 106–239.
[9] G. Booch, Object-Oriented Development, IEEE Transactions on Software Engineering SE-12 (2) (1986) 211–221.
[10] A. Fuggetta, E. Di Nitto, Software Process, in: Proceedings of the on Future of Software Engineering, FOSE 2014, ACM, New York, NY, USA, 2014, pp. 1–12. doi:10.1145/2593882.2593883.
[11] J. Coutaz, PAC: An Object Oriented Model for Dialog Design, in: Interact'87, Vol. 87, Elsevier, 1987, pp. 431–436.
[12] G. Calvary, J. Coutaz, L. Nigay, From Single-user Architectural Design to PAC*: A Generic Software Architecture Model for CSCW, in: ACM SIGCHI Conf. on Human Factors in Computing Systems, CHI '97, ACM, New York, NY, USA, 1997, pp. 242–249.
[13] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley Professional, 2013.
[14] G. E. Krasner, S. T. Pope, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, J. of object-oriented programming 1 (3) (1988) 26–49.
[15] D. M. Le, A Tree-Based, Domain-Oriented Software Architecture for Interactive Object-Oriented Applications, in: Proc. 7th Int. Conf. Knowledge and Systems Engineering (KSE), IEEE, 2015, pp. 19–24.
[16] D. M. Le, D.-H. Dang, H. T. Vu, jDomainApp: A Module-Based Domain-Driven Software Framework, in: Proc. 10th Int. Symp. on Information and Communication Technology (SOICT), ACM, New York, USA, 2019, pp. 399–406.
[17] M. Nosál', M. Sulír, J. Juhár, Language Composition Using Source Code Annotations, Computer Science and Information Systems 13 (3) (2016) 707–729.
[18] M. Fowler, T. White, Domain-Specific Languages, Addison-Wesley Professional, 2010.
[19] J. Gosling, B. Joy, G. L. S. Jr, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2014.
[20] A. Hejlsberg, M. Torgersen, S. Wiltamuth, P. Golde, The C# Programming Language, 4th Edition, Addison Wesley, Upper Saddle River, NJ, 2010.
[21] H. Wada, J. Suzuki, Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming, in: MODELS, LNCS 3713, Springer, 2005, pp. 584–600.

[22] V. Cepa, S. Kloppenburg, Representing Explicit Attributes in UML, in: 7th Int. Workshop on AOM, 2005.
[23] M. Sulír, M. Nosáľ, J. Porubän, Recording Concerns in Source Code Using Annotations, Computer Languages, Systems & Structures 46 (2016) 44–65.
[24] M. Balz, Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-Based Software Development, Ph.D. thesis, Universität Duisburg-Essen (Jan. 2012).
[25] D. Riehle, H. Züllighoven, Understanding and Using Patterns in Software Development, Theory Pract. Obj. Syst. 2 (1) (1996) 3–13.
[26] E. Gamma, R. Helm, R. Johnson, J. Vlissides, G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley Professional, Reading, Mass, 1994.
[27] D.-H. Dang, D. M. Le, V.-V. Le, AGL: A DSL for a Unified Domain Model, Tech. rep., VNU University of Engineering and Technology, Vietnam.
URL https://tinyurl.com/AGLTechnical
[28] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164.
[29] R. N. Thakur, U. Pandey, The role of model-view controller in object oriented software development, Nepal Journal of Multidisciplinary Research 2 (2) (2019) 1–6.
[30] V. Ajanovski, Integration of a course enrolment and class timetable scheduling in a student information system, International Journal of Database Management Systems 5 (1) (2013) 85–95.
[31] M. P. O'Mahony, B. Smyth, A recommender system for on-line course enrolment: An initial study, in: Proc. Int. Conf. Recommender Systems (RecSys), Association for Computing Machinery, 2007, pp. 133–136.
[32] Apache Causeway.
URL https://causeway.apache.org/
[33] OpenXava (2022).
URL http://openxava.org/
[34] OMG, Semantics of a foundational subset for executable uml models 1.5 (2021).
[35] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
[36] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Manifesto for Agile Software Development.
[37] S. Kent, Model Driven Engineering, in: M. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, no. 2335 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 286–298. doi:10.1007/3-540-47884-1_16.
[38] M. Brambilla, J. Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice, 1st Edition, Morgan & Claypool Publishers, 2012.
[39] E. Folmer, J. Bosch, Architecting for Usability: A Survey, Journal of Systems and Software 70 (1–2) (2004) 61–78.
[40] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, 2nd Edition, Packt Publishing, 2016.
[41] B. Combemale, O. Barais, A. Wortmann, Language Engineering with the GEMOC Studio, in: 2017 Int. Conf. Software Architecture Workshops (ICSAW), IEEE, 2017, pp. 189–191.
[42] B. Rumpe, K. Hölldobler, O. Kautz, MontiCore Language Workbench and Library Handbook, Shaker Verlag, 2021.
[43] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.
[44] S. Newman, Building Microservices: Designing Fine-Grained Systems, 1st Edition, O'Reilly Media, 2015.
[45] A. van Deursen, P. Klint, J. Visser, Domain-specific Languages: An Annotated Bibliography, SIGPLAN Not. 35 (6) (2000) 26–36.
[46] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.
[47] H. J. Köhler, U. Nickel, J. Niere, A. Zündorf, Integrating UML Diagrams for Production Control Systems, in: Proc. 22nd Int. Conf. on Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 241–251. doi:10.1145/337180.337207.
[48] I. A. Niaz, J. Tanaka, An Object-Oriented Approach to Generate Java Code from UML Statecharts, International Journal of Computer & Information Science 6 (2) (2005) 83–98.
[49] P. Selonen, K. Koskimies, M. Sakkinen, Transformations Between UML Diagrams, JDM 14 (3) (2003) 37–55.
[50] J. Warmer, A Model Driven Software Factory Using Domain Specific Languages, in: Model Driven Architecture- Foundations and Applications, Springer, 2007, pp. 194–203.
[51] J. Warmer, A. Kleppe, Building a Flexible Software Factory Using Partial Domain Specific Models (Oct. 2006).

Source Files (word or latex)

Click here to access/download

Source Files (word or latex)

latex.zip

Credit Author Statement

Duc-Hanh Dang: Conceptualization, Methodology, Writing- Original draft preparation, Investigation, Validation, Writing- Reviewing and Editing, Supervision

Duc Minh Le: Methodology, Writing- Original draft preparation, Software, Validation

Van-Vinh Le: Formal analysis, Software, Data curation, Validation

Declaration of interests

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

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: