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 works in DDD have been focusing on using annotation-based domain-specific languages (aDSLs) to build the domain model. However, within these works behavioral aspects, that are often represented using UML Activity and State machine diagrams, are not explicitly captured in the domain model.
Objective: This paper focuses on defining a novel unified domain modeling method in order to integrate behavioral aspects into domain models following the DDD approach. Specifically, behavioral aspects as part of a unified domain model are represented using a new aDSL, named activity graph language (AGL). Such an incorporation of the AGL and the previously-developed aDSL (DCSL) for a unified domain model would allow us to achieve three important features of a DDD: feasibility, productivity, and understandability.
Method: Our method consists in constructing a configured unified domain model within a domain-driven architecture. We used the annotation attachment feature of the host programming language like Java to attach AGL's activity graph directly to the activity class of the unified model, thereby, creating a configured unified model. The abstract and concrete syntax of AGL are also defined in this work. We demonstrate our method with a Java framework named jDomainApp and evaluate AGL using a case study to show that it is essentially expressive and usable for real-world software.
Results: This work brings out (1) a mechanism to incorporate behavior aspects for a unified domain model, in which a new aDSL named AGL is defined to represent the domain behaviors; and (2) 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

Object-oriented domain-driven design (DDD) [1] aims to iteratively develop software around a realistic model of the problem domain, which both thoroughly captures the domain requirements and is technically feasible for implementation. This requires a close collaboration among all the stakeholders (including domain experts, end-users, and developers), using a ubiquitous language [1] to construct a right domain model and resulting an object oriented implementation of this model. To achieve this, the DDD method tends to use a conceptual layered software architecture, which includes 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 (including user interface (UI), persistence, etc.) being realized in other layers surrounding this core. Recent works in DDD [2, 3] proposed annotation-based domain-specific languages (aDSLs), that are written inside a host object-oriented programming language (OOPL), to ease the construction of domain models. A straight-forward 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 of other concerns for a whole program. Another indirect way is to follow model-driven approaches by composing the domain model with other concerns expressed at either a high level using a general language like UML and DSLs. The final program then could be obtained by model transformations, either model-to-model or model-to-text ones. This work focuses on an alternative approach for this aim as a refinement of an aDSL-based software development method for DDD that we proposed in a recent work [4].

We aim to define an extension of domain model that allows us to represent behavior aspects of the domain within a unified model. The benefit of this extension is that we could obtain a composition of domain concerns for an executable version of the software at higher level of abstraction, which in turn significantly eases software construction from the domain model. However, it might be difficult for the modeler to represent behavior aspects within such a unified domain model. As a first step to get over this point is we define a mechanism with a language support to represent and incorporate the domain behaviors into the unified domain model. The proposed language would narrow the gap between the domain model and its implementation, thus it is easier to construct the software as a source model of transformations from a domain model incorporated with behavior models (in UML and DSLs).

Specifically, we define a novel aDSL, named AGL (Activity Graph Language) with two main aims: (1) to represent behavioral aspects (that could be captured using UML Activity diagrams and Statecharts [5]) and (2) to incorporate them as part of the unified domain model. For the first aim, we scope AGL around a restricted domain of the UML activity graph language that is defined based on essential UML activity modeling patterns [5]. We adopt the meta-modeling approach for DSLs [6] and use UML/OCL [5, 7] to specify the abstract and concrete syntax models of AGL. For the second aim, we employ our previously-developed aDSL, named DCSL, in order to express the unified domain model. The unified model is viewed as an extended domain model in MOSA (a module-based software architecture [8] that we have recently developed for DDD). This model 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, that performs a set of core actions on the software modules in MOSA. These actions concern the manipulation of instances of the domain class (owned by the corresponding software module). We demonstrate our method with an implementation in jDomainApp and evaluate AGL to show that it is essentially expressive and usable for designing real-world software.

In brief, our paper makes the following contributions:

- A mechanism to incorporate behavior 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 **??** discusses 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 is that (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 so-called the 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 works on DDD.

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

In practical software development, the MVC architecture models are adopted so that the software can have some sort of GUI to assist the development team in constructing it. The main reason for this is rooted in a general understanding (at least up to recently) that software construction can not be fully automated [10], due primarily to the human factors that are involved in the development process. Software that is designed in MVC consists of three components: model, view, and controller. The internal design of each of the three components is maintained independently with minimum impact on the other two components. Modularity can further be enhanced by applying the architecture at the module's level (e.g. , by adopting another agent-based design architecture named PAC [11]), thereby creating a hierarchical design architecture in which a software is composed of a hierarchy of software modules. A software *module* (called PAC object in [11] and, more generally, agent in [12]) is a realization of a coherent subset of the software's functions in terms of the architectural components.

To construct DDD software from the domain model requires an architectural model that conforms to the generic layered architecture [1, 13]. A key requirement of such model is that they position the domain model at the core layer, isolating it from the user interface and other layers. Evans [1] suggests that the MVC architecture model [14] is one such model. The existing DDD frameworks [2, 3] support this suggestion by employing some form of MVC architecture in their designs. We observe from all of these works that the user interface plays an important role in presenting a view of the domain model to the stakeholders in such a way that help them to effectively build the domain model. We thus argue that the MVC architecture must be the backbone of any DDD tool that conforms to the DDD's layered architecture.

Our previous works [8, 15] proposed a variant of the MVC architecture for DDD software, called **module-based software architecture** (**MOSA**). A key feature of this architecture is that it supports the automatic generations of software modules from the domain model and of the software from these modules. A **MOSA model** consists in a set of MVC-based module classes. A **module class** is an MVC-based structured class [**?**] that represents modules. This class is composed of three components: a domain class (the model), a view class (the view) and a controller class (the controller). The module class becomes the *owner* of the model, view and controller. The view and controller are parameterized classes that are created by binding the template parameters of two library template classes, named `View` and `Controller` (*resp.*), to the domain class. We present in [8] a technique for semi-automatically generating a module class from the domain class that it owns. Further, the view is designed to reflect the model structure. A set of module classes are used as input for the jDomainApp software framework [16] to automatically generate software. In this paper, we will assume that a module class is defined for every domain class.

Example. Figure 1 shows five module classes of course management domain (CourseMan). The parameter bindings are depicted by dashed lines, whose `Controller`'s and `View`'s ends are drawn with the symbol '◯'. For example, the module class `ModuleStudent` is composed of three component classes: the domain class `Student`, the view `View`⟨`Student`⟩ and the controller `Controller`⟨`Student`⟩.

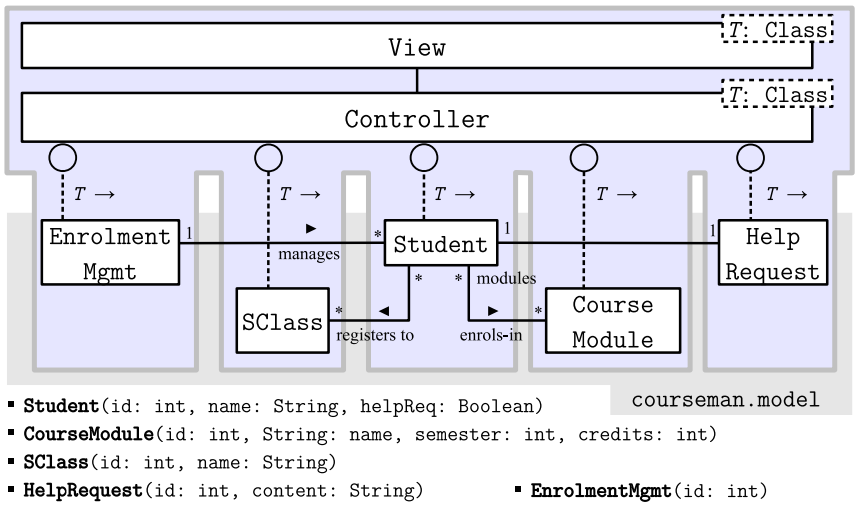


Figure 1: The MOSA model of CourseMan.

We argue that MOSA captures the essence of object-oriented software design in a modular, MVC-based design structure. According to Booch [9], an object-oriented software consists of objects and their interactions that are realized though behavior invocation. Given that the domain model is expressed in DCSL (further explained in SubSect. 2.3), the MOSA model that has this model at its core helps produce software that possesses the essential behaviors. First, objects are instances of the domain classes in the domain model, which are represented in DCSL with the essential structural features. Second, interaction among the objects of a group of domain classes is performed through an event-based message passing mechanism that is managed by the owner modules of these domain classes. This mechanism, which is described in detail in [16], maps events to the essential behaviors that are supported in DCSL. The events can be triggered by the user interaction on the 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 an OOPL's abstract syntax model [4] that consists of the following meta-concepts: class, field, method, parameter, annotation, and property. These meta-concepts are common to two popular host OOPLs: Java [19] and C# [20]. Our idea of using annotation to represent modeling rules and constraints is inspired by AtOP [21–24]. In principle, AtOP extends a conventional program with a set of attributes, which capture application- or domain-specific semantics [22]. These attributes are represented in contemporary OOPLs as annotations. We stated in [4] that using aDSL for DDD brings three important benefits for domain modeling: feasibility, productivity, and understandability. Feasibility comes from the fact 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 that we developed to express domain models. A key feature of DCSL is that its meta-concepts model the generic domain terms that are composed of the core OOPL meta-concepts and constraints. More specifically, meta-concept **Domain Class** is composed of meta-concept `Class` and a constraint captured by an annotation named `DClass`. This constraint states whether or not the class is mutable. Similarly, meta-concept **Domain Field** is

composed of meta-concept `Field` with a set of state space constraints. These constraints are represented by an annotation named `DAttr`. Meta-concept **Associative Field** represents Domain Field that realizes one end of an 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, meta-concept **Domain Method** is composed of `Method` and commonly-used constraints and behavior types that are often imposed on instances of these meta-concepts in a domain model. The essential behavior 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.

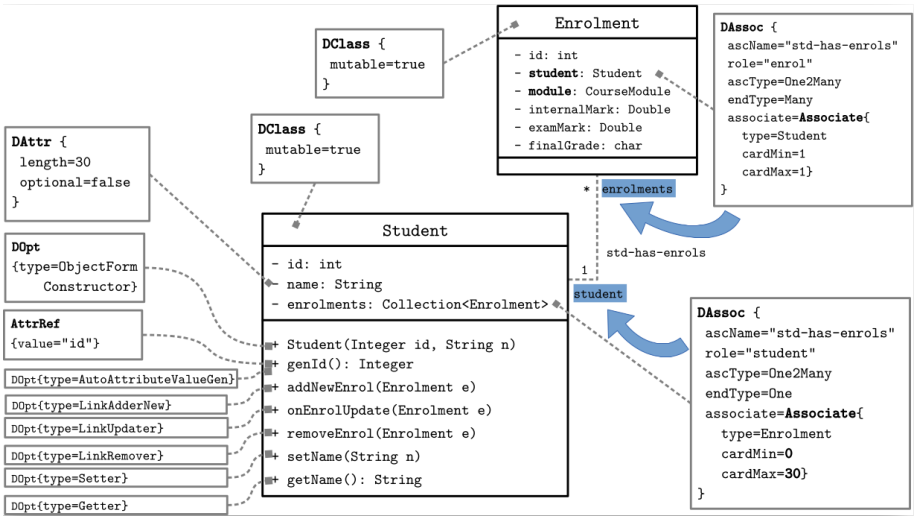


Figure 2: A partial CourseMan domain model expressed in DCSL (adapted from [4]).

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. 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 an optional domain field, whose maximum length is 30 (characters). 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 behavior type. For instance, method `genId`, whose behavior 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 shows an essential domain model for CourseMan, that 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:

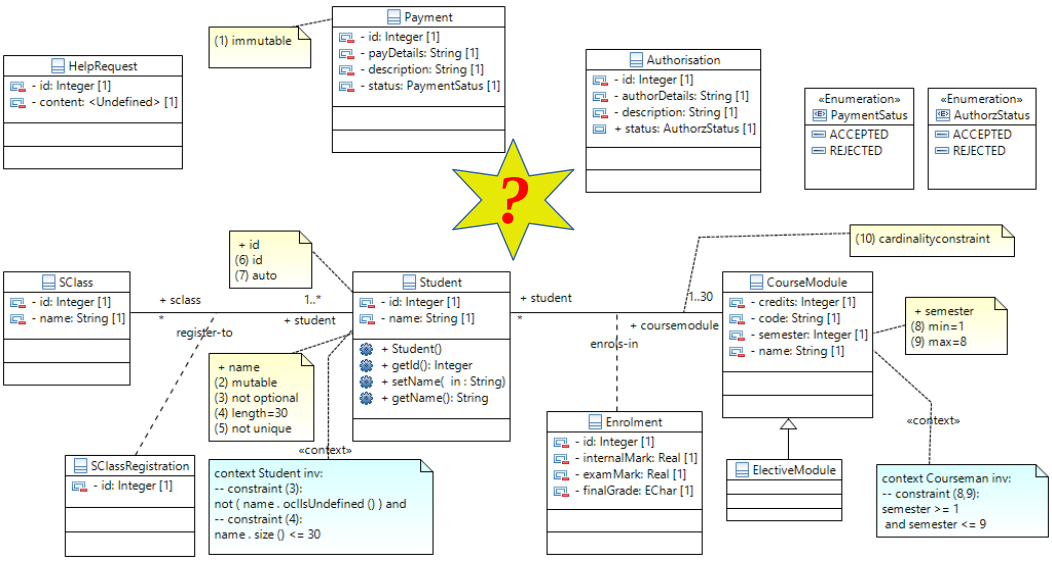


Figure 3: The essential domain model of CourseMan.

- `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.

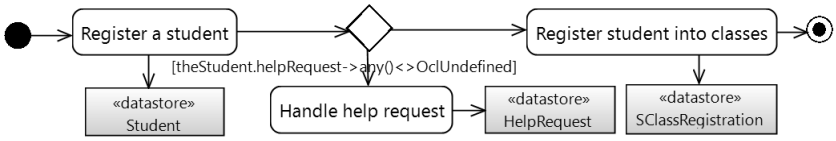


Figure 4: A UML Activity diagram to represent the enrolment management activity.

Figure 4 shows a UML Activity diagram for the enrolment management activity. This activity involves registering `Students`, enrolling them into `CourseModules` and registering them into `SClasses`. In addition, it would allow a `Student` to raise a `HelpRequest` during the enrolment process. We might consider the domain behavior as a new concern that needs to be composed with the essential domain model, as shown in Figure 3, for an executable version of the software. Since this domain behavior is currently captured in UML, we would need a further mechanism to maintain a consistency between the two models, toward composing them, normally at an implementation level. As an alternative approach for this aim, following the DDD approach introduced in our previous work [4], we would consider such a behavior concern as an extension of the essential domain model for a unified domain model (i.e., a DDD with the key features, feasibility, productivity, and understandability as explained above). To achieve the goal we face two main challenges that motivates this work as follows:

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 explains our basic idea of incorporating behavior aspects as part of a unified domain model.

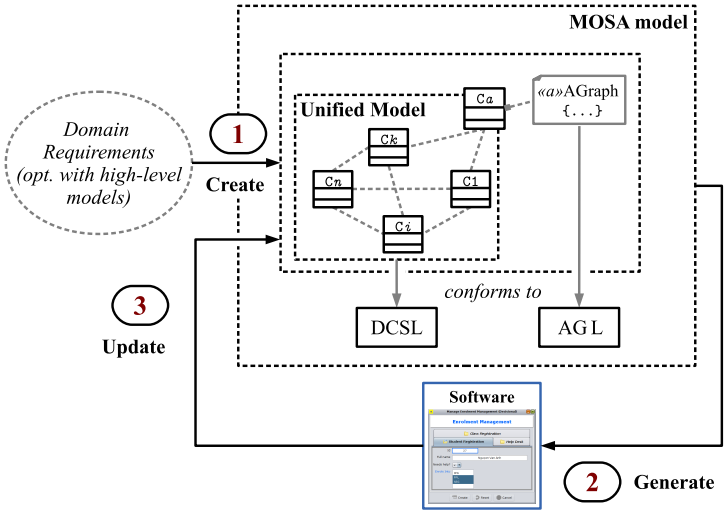


Figure 5: An overview of our method.

3.1. Basic Idea

Figure 5 overviews our proposed method. This method conceptually consists in iteratively performing three steps. First, we take as input domain requirements that are captured by an essential domain model (for a structural view including domain concepts and relationships) together with domain behaviors (specified as a behavioral view with UML Activity diagrams). 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. The former model (the DCSL model) is an extension of the essential model to relate this structure view with the behavior view. A detailed definition of the unified model is provided in Section 3.3. For the latter model (the AGL model), wee need to define AGL as a new aDSL to capture a semantics domain of module actions for the domain behaviors. A detailed explanation of this point is provided in Section 3.2 and Section 4. Second, the unified model composed with the AGL model is taken as input to automatically generate a GUI- and module-based software. This software is presented to the domain expert in order to get feedback. Third, if there is feedback, then the input model will be updated and the cycle continues. If, on the other hand, the domain expert is satisfied with the models, then the cycle ends.

3.2. Incorporating Domain Behaviors

We introduce a mechanism to incorporate domain behaviors into a domain model. The mechanism is defined based on the structure and behavioral semantics of MOSA at two points. First, each module class that owns a corresponding domain class is defined with a set of essential actions (i.e., *atomic actions* as explained in Section 4) in order to manipulate the instances of the domain class. Second, domain behaviors are considered as collaborations among modules in MOSA: Each module collaboration is on the one hand

coordinated by a composite module (in MOSA), on the other hand, captured by a corresponding activity model. Specifically, we map each of the activity models, e.g., the enrolment management in CourseMan as depicted in Figure 4, to a new domain class (referred to as a so-called activity class that is owned by a corresponding activity module, e.g., the `ModuleEnrolmentMgmt` in CourseMan). The containment tree of the composite module allows promoting it as the main module for managing the entire activity.

Within the proposed mechanism, basically, we could employ UML Activity diagrams to represent domain behaviors but we need to restrict them for a semantics domain corresponding to the behavior semantics of the composite module (coordinating a collaboration among moudules). To define such a semantics domain we employ a pattern-based approach: Domain behaviors are specified using UML Activity diagram with basic constructs corresponding to the five essential activity modeling patterns as presented in [4]. We named the patterns after these five elementary activity flows: *sequential*, *decisional*, *forked*, *joined* and *merged*. Further explanation for this point is provided in Section 5.

3.3. Unified Model

A *unified class model* is an extended domain model for incorporating domain behaviors. As explained above, the domain behaviors within our approach are captured as activity models with UML Activity diagrams. Within the extension we newly add so-called *activity classes*, e.g., class $\mathtt{C_a}$ in Figure 5, for each activity of the domain behaviors. The activity class is referred to and handled by a corresponding activity model (that can be seen as an activity graph). Thus, the behavioral logic of the activity is realized and synchronized with current states of the domain model. Such a unified class model could be realized in DCSL, and we refer to the resulting DCSL model as a *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 **unified model** is a DCSL model that realizes an 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 $c_1, \ldots, c_n$.

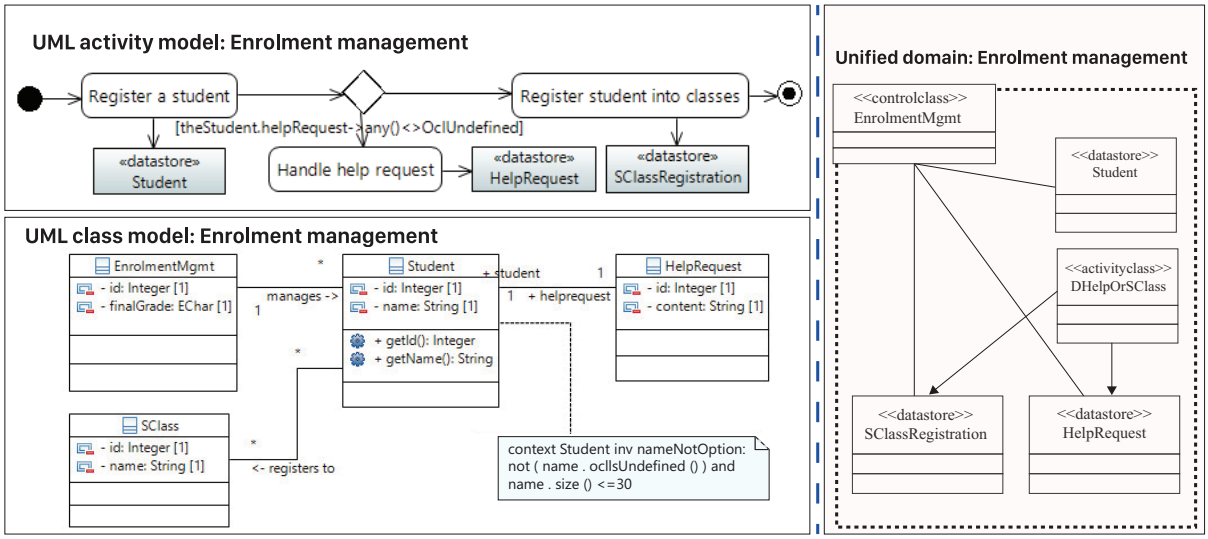


Figure 6: (A: Left) The UML activity and class models of a CourseMan software variant that handles the enrollment management activity; (B: Right) The unified model that results.

Example. To illustrate, Figure 6(A) shows the UML activity and class models of a CourseMan variant that handles the enrollment management activity. In this variant, students are allowed to request help after the initial registration. The accompanied class model is extracted from the CourseMan's conceptual model as shown in Figure 3. Figure 6(B) shows the resulting unified model of the activity. This model consists of five domain classes and realizations of five activity-specific associations. To ease reading, we omit the domain-specific associations that are shown in the UML class model in Figure 6(A). Class `EnrolmentMgmt` is the activity class. Class `DHelpOrSClass` is a decision class, which captures the domain-specific decision logic. The remaining three classes are data classes that realize the three data stores. These data classes also correspond to three domain classes in the UML class model.

Among the five associations, three associate `EnrolmentMgmt` and the data classes. These associations are used to bind the modules of these data classes to the containment tree of `ModuleEnrolmentMgmt`. The remaining two associations associate the decision class `DHelpOrSClass` to two data classes (`SClassRegistration` and `HelpRequest`), which realize the data stores connected to the two action nodes branching of the decision node. These associations are weak dependency associations and only added in this case because the decision logic encapsulated by `DHelpOrSClass` needs 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 recursively 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 behavior 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` $= s$.* □

Note the following about the above definition. First, we use module states to abstract from the local pre- and post-conditions of each action. This abstraction enables us to flexibly combine actions based on states to construct more complex ones. A **module state** abstracts from 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, resulting in what we call **concurrent state**s. We write these states using the operator '+'. The `postStates` of primitive action consists of a single state, while that of more complex actions (discussed in Section 4.3) consists of multiple states.

Second, because each action concerns manipulating the values of some domain fields of the domain class, the action inputs, if any, need to be those that are used for updating these fields. Thus, we define action inputs as a (possibly empty) field-value set. An element of this set is a pair (f, v), where f is a field name and v is a value. The value v in each pair is either specified by the user or from another action that has previously been performed. The latter case occurs when we compose actions together to form more complex behavior. We will explain action composition in the subsequent subsections.

Third, the action output consists of at most one type, which is the domain class of the current module. Further, 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`,

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).

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 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 with an interaction scenario. We model this sequence using structured action of 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.postStates $\subseteq$ a_{i+1}.preStates $(\forall a_i, a_{i+1} \in S)$. *$S$ has the following properties:*

- S.preStates = first(S).preStates
- S.postStates = last(S).postStates
- S.fieldValSet = first(S).fieldValSet
- S.output = last(S).output □

Example. Figure 7 shows an ASE that creates a new domain object whose type is the domain class T of a module. This ASE consists in a sequence of four atomic actions and is characterised 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, so that interested listeners of this event can handle. This

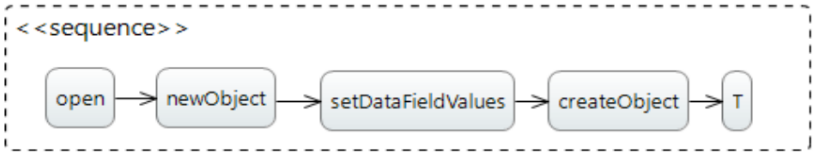


Figure 7: An ASE that creates a new domain object of a module's domain class (typed T).

action then leads to the execution of the second atomic action: `newObject`. This sequence is valid because, as listed in Table 1, `open.postStates` $\subset$ `newObject.preStates`. Action `newObject` 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. Finally, because `setDataFieldValues.postStates` $\subset$ `createObject.preStates` we place `createObject` as the next (and final) action of the ASE. This action creates a new domain object (using values of the view fields).

A useful property that emerges from our notion of ASE is that there exists a natural multi-level nesting of ASE-backed behaviors along a path in the module containment tree. More specifically, an ASE S is '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 which is performed on the child module of type `ModuleEnrolment`. The ASE of `ModuleStudent` itself is nested inside that of `ModuleSClass`, thereby creating a 2-level nesting. The definition of ASE gives rise to the notion of *reachable state*, which is a module state that is reachable from a given action. We discuss this notion 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, the post-state of an atomic action is reachable from its own action. Let us define the reachable states of atomic actions shown in Table 1. First, the reachable states of 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 raised only once. Second, the reachable states of `newObject` include `NewObject`, `Editing`, `Created`, `Reset`, and `Cancelled`. The action `newObject` additionally cannot reach `Updated` and `Deleted`. This is because this action is reserved for creating a new object. It thus cannot also lead to updating or deleting an existing object. Third, the reachable states of action `setDataFieldValues` include `Editing`, `Created`, `Updated`, and `Reset`. The action `setDataFieldValues` cannot reach `NewObject`, `Deleted` and `Cancelled`. This is because this action concerns only input data and thus cannot initiate or cancel object creation, nor can it lead to object deletion. Finally, with the remaining five actions each has only one reachable state, which is their own states. 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 us consider the SAA $(\texttt{newObject}, \{\texttt{Created}, \texttt{Cancelled}\})$, which represents a common ASE set that starts with the action `newObject` and ends only when either the state `Created` or the state `Cancelled` is detected. The ASE set consists of the following frequently-occurring ASEs. The first ASE is the one described earlier in Figure 7 but excludes the first action. We assume here 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 focuses on explaining our pattern-based approach for incorporating domain behaviors into a domain model. Each domain behavior, that could be captured at a high-level description using an UML activity diagram together with domain-model based statements, is translated into a specification with two parts: (1) part of the unified 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 model. We realize such a translation by applying a domain behavior pattern. The first catalog of domain behavior patterns is defined corresponding to the five essential UML activity modeling patterns [4], 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 model and a **template configured unified model** that realizes it. The template model is a 'parameterized' configured unified model, in which elements of the non-annotation meta-concepts are named after the generic roles that they play. For brevity, we will 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 model).

We would illustrate each pattern of our first catalog with a variant of the unified model for the enrolment management activity of CourseMan. Due to the limitation of the length of this paper, we focus on the *Decisional Pattern* to illustrate the approach. For a detailed explanation of the four remaining patterns, including *Sequential Pattern*, *Forked Pattern*, *Joined Pattern*, and *Merged Pattern*, we refer the reader to the technical report [27] for this paper.

The top-left of Figure 8 shows the UML activity model, 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 that is referenced by the control node c_k of the activity model. Class `D` is a decision class, which 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 would need none or some of these associations.

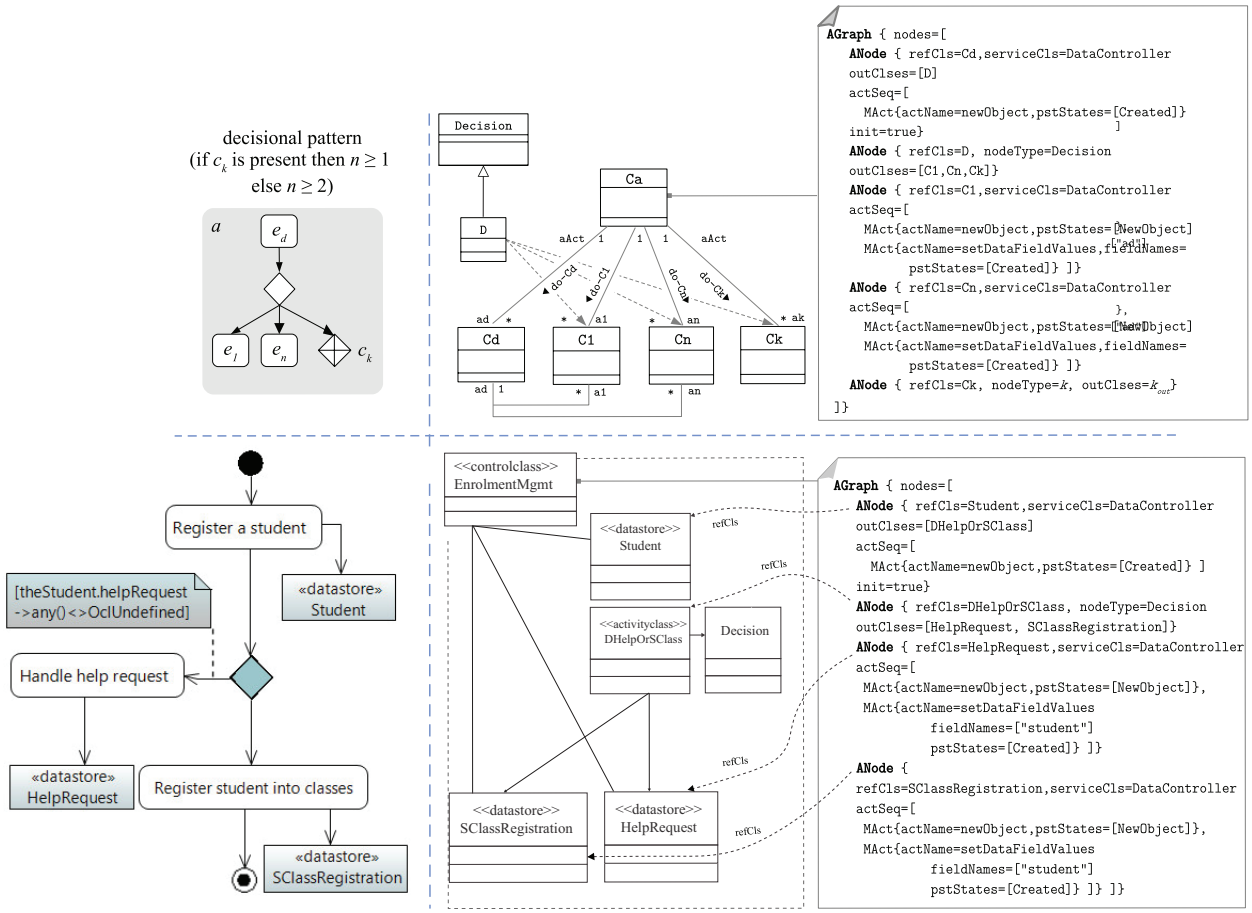


Figure 8: The decisional pattern form.

Class `Ca` has one-many associations to the other four domain classes. Note that the association to `Ck` can be used as a bridge in a larger activity model to other activity flow blocks. This association is applied differently if c_k is a decision node. In this case, `Ck` has no associations and thus the association to `Ck` is replaced by (or "unfolded" into) a set of associations that connect `Ca` directly to the domain classes of the model containing `Ck`. In the template model, the two associations between `Cd` and `C1`, `Cn` reflect the fact that both `C1` and `Cn` know about `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 consists of five `ANode`s. The first `ANode` is to create a new `Cd` object. The second `ANode` is to run the decision logic. The third and fourth `ANode`s represent the two decision cases: the first results in creating a new `C1` object for the specified `Cd` object, the second, which is repeated for all n, results in creating a new `Cn` object for the same `Cd`. The fifth `ANode` is used for the case that `Ck` is specified. It uses two variables k and k_{out}, both are dependent on `Ck`. Variable k specifies the control node type, while variable k_{out} specifies the array of output domain classes of `Ck`.

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

Figure 9 shows the three GUI snapshots of the example. The first GUI is for student registration. The second and third GUIs are for the cases that help request is and is not requested (*resp.*). The activity's GUI contains the GUIs of the three actions in separate tabs. Under both cases of the decision, the `Student`

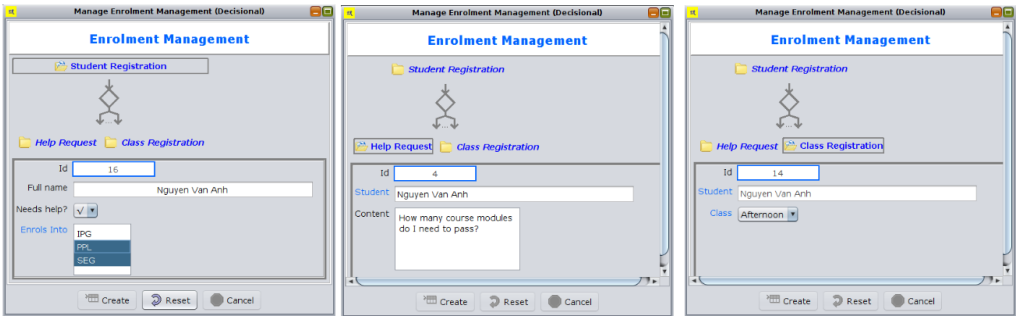


Figure 9: The decisional pattern form view of enrolment management.

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 presented 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 proceed three main steps as follows:

1. Representing the input UML Activity diagram and the template configured unified model of the pattern as activity graphs;
2. Defining the matching of the template on the activity in order to define ANodes, their sequence, and domain classes referenced by them. The reference from ANodes to domain classes (expressed with the keywords `refCls` and `outCles`) provide a mapping between the input activity and the unified model;
3. Defining the generic behavior (expressed by a SAA with the keywork `actSeq`) of the domain module *w.r.t* the domain class referenced by each ANode. This helps synchronize the execution state of the activity and the current state of the unified model.

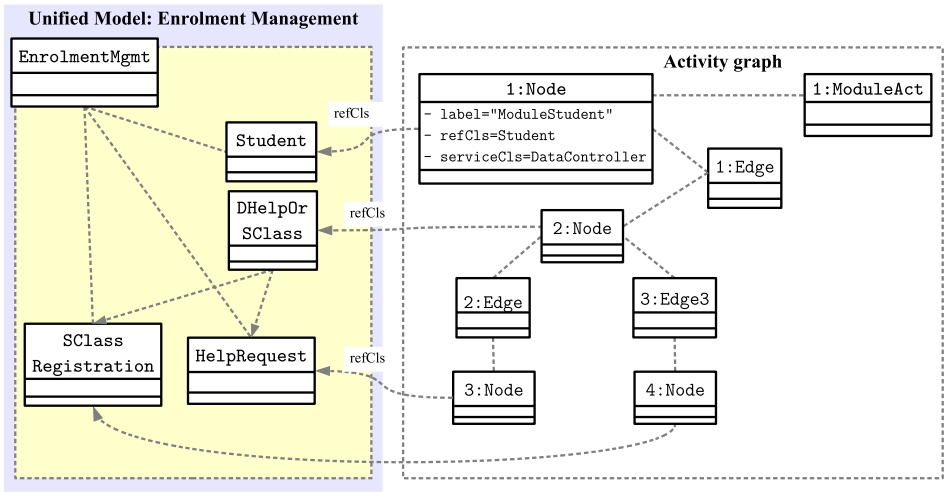


Figure 10: (LHS) A repeat of the unified model shown in Figure 6; (RHS) The activity graph of this model.

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"}

Example. The right-hand side of Figure 10 is an activity graph of the enrollment management activity of the CourseMan software variant introduced earlier in Section 2. The left-hand side of the figure is the corresponding unified model of this activity, which is repeated from Figure 6 to show links with the activity graph. 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 `Node`s 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`'s value of each node is depicted in the figure by a dashed curve (labeled "`refCls`") that connects the node to the referenced domain class in the unified model.

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 combination of a unified model and an activity graph. It highlights the sub-set of modules that owns the activity classes and how these modules trigger the execution of the activity graphs of the associated activities.

Definition 7. *Given a unified model D that contains a non-empty set of activity classes, each of which is attached to an AGL specification describing the activity graph logic of an activity of the domain. 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 briefly specify AGL as an aDSL for incorporating domain behaviors into a domain model. The language is used to create activity graphs by configuring them directly on the domain model using annotations. Adapting the meta-modeling approach for DSLs [6], we focus on defining an *abstract syntax meta-model* (ASM) and an annotation-based textual *concrete syntax model (CSM)* for AGL. We only explain the syntactic aspects of AGL since its behavior semantics is characterized in Sections 4 and 5.

6.1. Abstract Syntax

We describe the AGL's domain requirements in terms of the following inclusion (I), exclusion (X) and restriction (R) clauses that are applied to the UML activity graph requirements [5]: (I1) module action (described in Section 4) as a special form of action; (R1) executable node performs a sequence of module actions; (R2) value specification [5, p. 374]) is only applied to decision node; (X1) using variable with activity ([5, p. 417]); (X2) variable action [5, p. 467]; (X3) activity edge [5, p. 373] is without guards.

I1 and R1 are needed to incorporate the activity graph into MOSA. R2 is a safe restriction because, according to the specification, value specification is mainly used for specifying conditions on decision nodes. X1 and X2 concern the use of variables. According to the UML specification, variable is an alternative to using object flow. The exclusion of edge guards (X3) is just a deliberate omission at this stage since we would focus on supporting the core structure of the activity graph. We plan to remove X3 in future work.

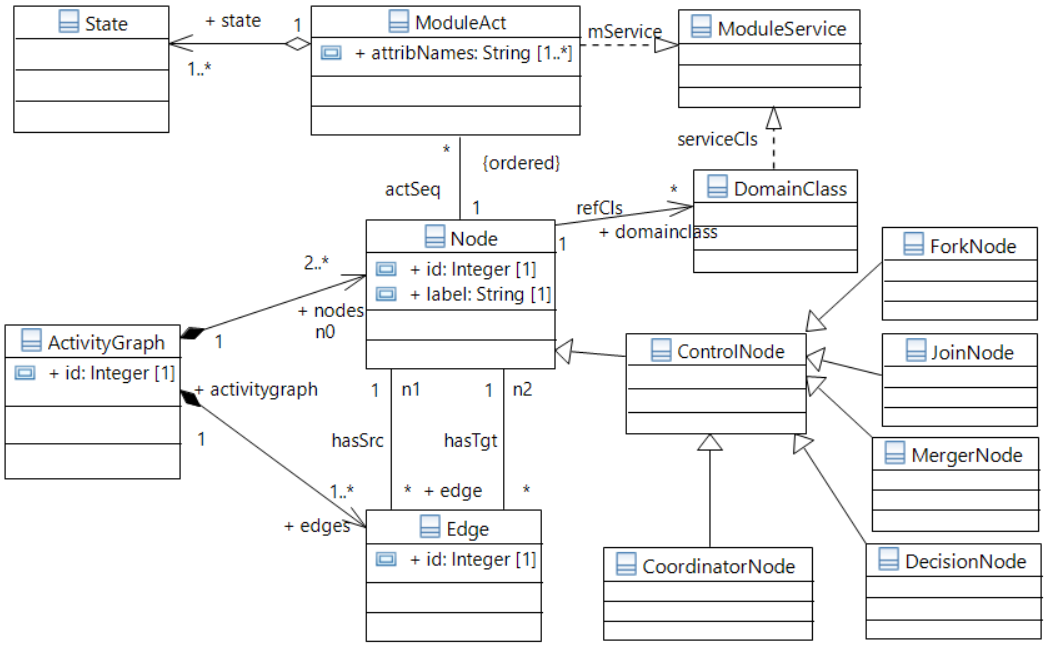


Figure 11: A simplified metamodel for AGL's abstract syntax.

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

Meta-concept `Node` represents activity nodes and has four properties. Attribute `label` realizes the node label. Property `out` is derived from the association `hasSrc(Edge,Node)`, which records the outgoing edges from a `Node`. The next two properties specify the referenced module that is referenced by this node. Specifically, property `refCls` references the domain class of the *ref* module. Property `serviceCls` specifies the actual `ModuleService` class of the module. It is through a module service object of `serviceCls` that the current `Node` is able to perform the `ModuleAct`s specified by the property `actSeq`. This property corresponds to the target role of the association from `Node` to `ModuleAct`. Meta-concept `CoordinatorNode` is used to represent a task group. It does not pass its data out to the outgoing edges. Instead, it passes the original input data out. The coordinator's UI serves as the container of those of the member tasks, so that the user can have a whole picture of the group. The member tasks themselves interact with each other to perform the group's logic. An example for this meta-concept is shown 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).

TODO: We need to provide well-formedness (WFRs) rules for the metamodel to obtain valid AGL models: (1) WFRs from UML metamodel for valid activity graphs; (2) WFRs to ensure AGL models obtained as a

composition of available domain behavior patterns.

Due to limited space, the well-formedness OCL rules for this model are only shown in the technical report [27] for this paper.

6.2. Annotation-Based Textual Concrete Syntax

We aim to define an annotation-based textual concrete syntax for AGL. To obtain this goal, we define a metamodel for the concrete syntax (CSM) of the AGL by a transformation from the abstract syntax ASM. Figure 12 shows the CSM that takes the annotation-based form, suitable for being embedded into a host OOPL. Furthermore, we will strive for a compact CSM that uses a small set of annotations. From a practical standpoint, such a model is desirable since it will result in a compact concrete syntax, which requires less effort from the language used to construct a unified domain model.

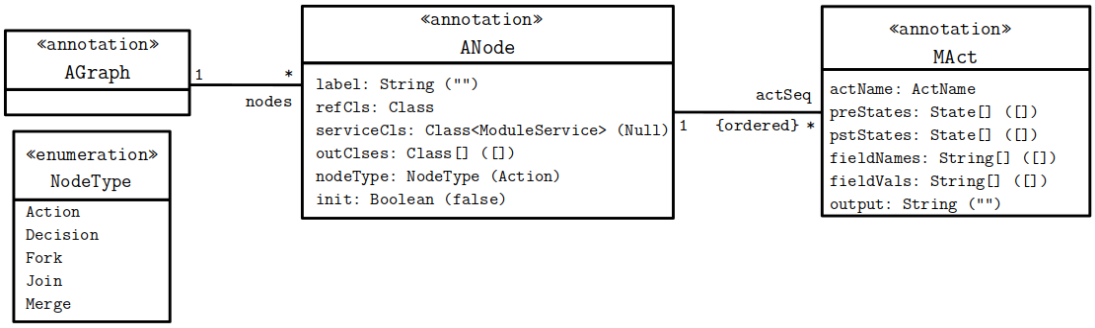


Figure 12: Illustration for AGL's annotation-based textual concrete syntax, realized in Java.

To realize the transformation, we proceed in two steps. First, we transform ASM into the meta-model CSM_T that is compact and suitable for annotation-based representation. CSM_T consists of three meta-concepts: activity graph (`AGraph`), activity node (`ANode`), and module action (`MAct`). Second, we transform CSM_T into the actual annotation-based CSM that is "embedded" into OOPL. This transformation is required because although CSM_T is suitable for OOPL's representation, it is still not yet natively in that form. This CSM is constructed from the following three OOPL meta-concepts that were discussed in Section 2.3: `class`, `annotation`, and `property`. Because CSM is embedded directly into OOPL, its structure helps define the core structure of a CSM model of the AGL's textual syntax. Adapting the concrete syntax meta-modeling approach [6] to AGL, we argue that its CSM will contain, in addition to the above core, meta-concepts that help describe the structure of the BNF grammar rules. 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 would refer the reader to Appendix B of the technical report [27] for this paper.

Example. The bottom-right of Figure 8 depicts the configured unified model of CourseMan's enrolment management activity. In this figure the entire AGL specification is defined by an `AGraph` element, which is written within a note box attached to the activity class `EnrolmentMgmt` of the unified model. For exposition purposes in this paper, we will textually write an AGL specification using the structured note box notation of DCSL, as explained in Section 2.3.

7. Case Study and Tool Support

This section first presents a relatively complex case, order management domain (OrderMan), that is adapted from the OMG/UML specification [5, p. 396]. The aim is to illustrate the usability of AGL and to investigate how our proposed method is applied to develop software for a real-world problem domain. This section then presents tool support, with a focus on explaining main design choices and used technologies.

7.1. Case Study: OrderMan

Figure 13 shows the UML activity diagram for the process in OrderMan to handle orders. To obtain the unified domain model for OrderMan, as shown in Figure 14, we have applied all the domain behavior patterns of our first catalog. Specifically, Figure 14(A) represents an activity graph with nodes labeled with numbers for the activity HandleOrder. It also represents part of the unified class model (for a full version, we refer the reader to the technical report [27]), that consists of one activity class (`HandleOrder`), four main data classes (`CustOrder`, `Invoice`, `Shipment`, `Payment`), four control classes (`AcceptOrNot`, `Delivery`, `CompleteOrder`, `EndOrder`), and four remaining classes *w.r.t* coordinator nodes (`FillOrder`, `CollectPayment`, `ShipOrder`, `AcceptPayment`). As explained in Section 6, coordinator nodes are used to provide a whole picture of a task group. For example, the coordinator node Fill Order (node 3) handles the following two tasks: Update Order (node 5) and Delivery Order (node 6). Fill Order simply coordinates the tasks, ensuring that Update Order is performed first then Delivery Order. It does not contribute any data to this flow. However, it enables the user to observe and perform the task flow on the UI.

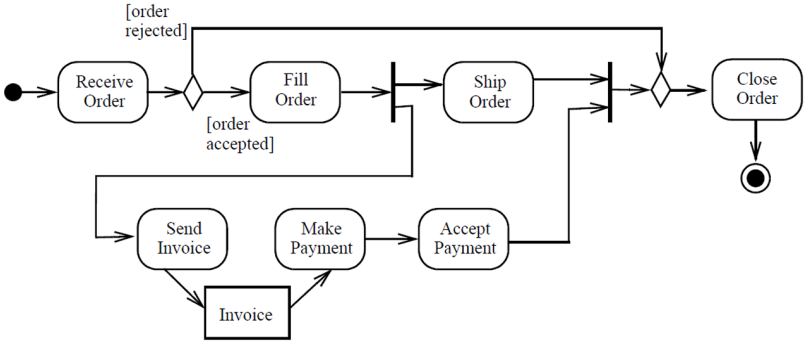


Figure 13: The UML activity diagram for the process to handle orders, adapted from [5, p. 369].

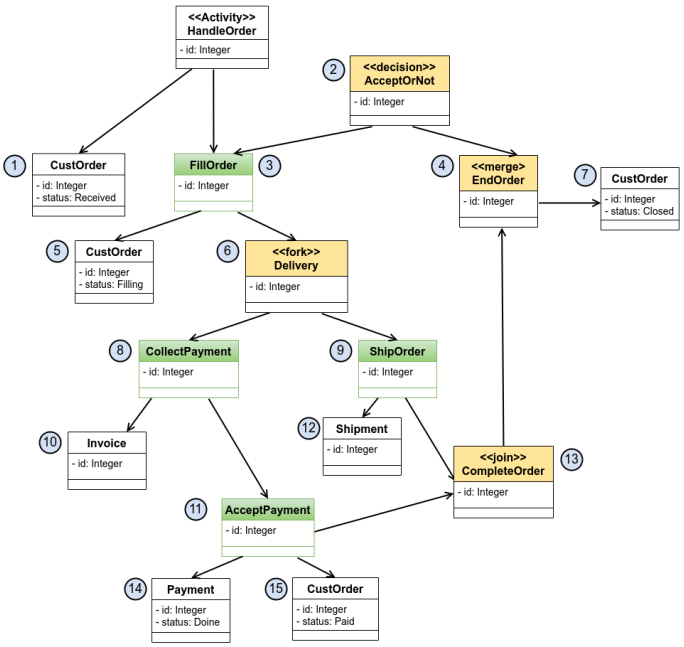
Figure 14(B) highlights for each node of the activity graph (1) the mapping from the node to a corresponding module (*w.r.t* the domain class referenced by the `refCls`), (2) the out nodes (*w.r.t* the `refCls`), and (3) the `ModuleAct` objects each of which specifies a SAA for the behavior of the module. These `ModuleAct` objects together with SAAs are listed in Figure 14(C). In order to obtain an OrderMan software with the GUI, as shown in Figure 15, the unified model incorporating the AGL specification (for the activity model) needs to be encoded in Java. The implemenation for OrderMan is available at the git repository[1]. Section 7.2 provides a detailed explanation of this implementation.

7.2. Tool Support

As illustrated in Figure 16, we realized our method with a support tool based on JDomainApp, a Java software framework that we reported in previous works [4]. The tool is available at the git repository[2]. Conceptually, the tool consists of three key components: model manager, view manager, and object manager. First, the **model manager** is responsible for registering the configured unified model and making it accessible to other components. The left window in Figure 16 depicts the list of module classes (e.g., `ModuleHandleOrder`) and corresponding domain classes (i.e., `HandleOrder`). As an activity domain class the `HandleOrder` is attached with an annotation `@AGraph` for the AGL specification, as presented in the middle window in Figure 16. The annotations to realize AGL are defined by the jdomainapp framework with the java projects as shown in the left window of this figure.

Second, the **view manager** is responsible for (1) automatically generating the entire GUI of the software from the unified model and (2) for handling the user interaction performed on this GUI. The GUI consists

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



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`.

of a set of object UIs (one for each module's view), and a desktop for organising these UIs. The two tasks basically are based on the configuration description of modules that is specified in MCCL [4]. For example, the module class `ModuleHandleOrder` that owns `HandleOrder`, as presented in the right window of this figure, is specified together with a module description in MCCL.

Third, the **object manager** is responsible for managing the run-time object pool of each domain class and for providing a generic object storage component for storing (retrieving) the objects to (from) external storage. As of this writing, the tool supports both file-based and relational database storage. The relational data model is automatically generated from the unified model the first time the software is run.

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();
}
```

In the rest of this section, we would further explain how a module class *w.r.t* an activity class can handle the execution of the activity graph *w.r.t* its AGL specification. Let's focus on a concrete situation with the `ModuleHandleOrder` module with its activity class, `HandleOrder`, realized as in Listing 1. When

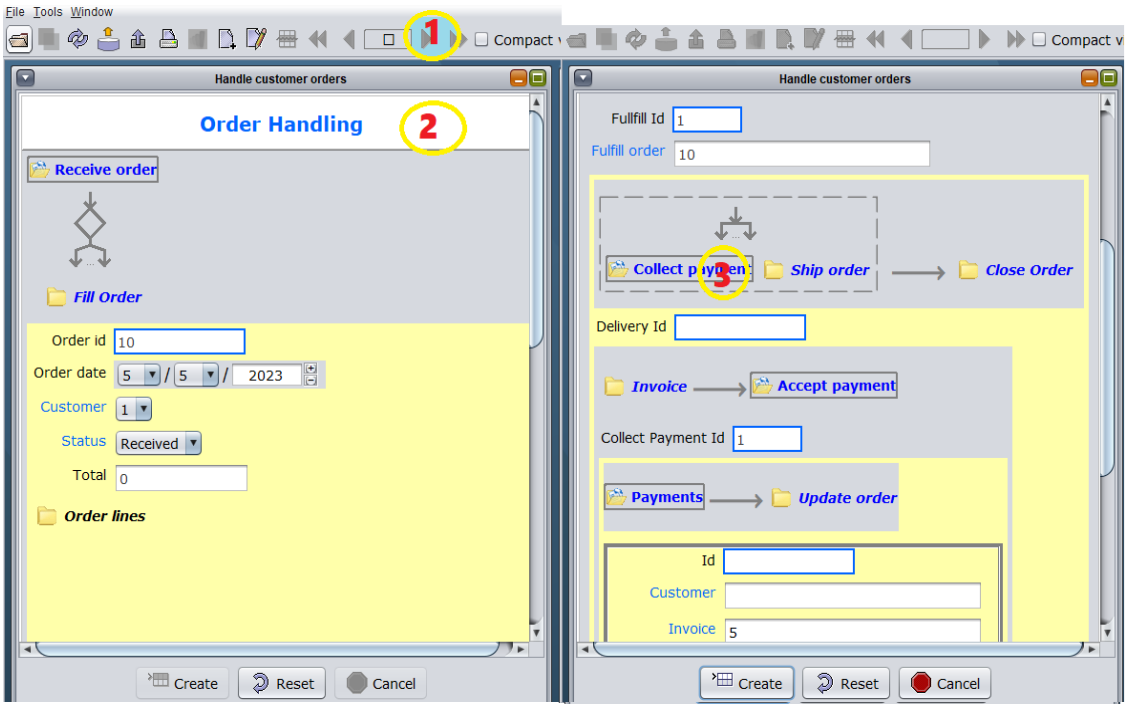


Figure 15: The GUI of the OrderMan software generated by the tool.

the software runs, an instance of the `ModuleHandleOrder` module is invoked. Based on the configuration description of this module as shown in the right window of Figure 16, an activity model (`ActivityModel`) as a composition of the `HandleOrder` object (owned by the `ModuleHandleOrder` module) and an activity graph *w.r.t* the AGL specification of the activity class `HandleOrder` is created. The activity diagram could be defined because the `HandleOrder` object can be considered as an `AGrap` object (based on the annotation mechanism in Java). The definition of the annotation `AGraph` is shown as in Listing 2. The `AGraph` object allows each of its `ANode`, e.g., the `ANode` *w.r.t* node 14 as shown in Listing 1, as well as the domain class referenced by the `ANode` (i.e., the `Payment` class) could be handled by the `ModuleHandleOrder` module.

8. Evaluation

Reviewer: Section 8. The evaluation is very weak. Firstly, there are no research questions. Second, the expressiveness of AGL is argued but not evaluated. To demonstrate expressiveness, the paper could report on a couple of realistic case studies showing the extent to which AGL can capture all the necessary behaviour without resorting to code. For example, the approach seems limited to CRUD applications, so it is not clear whether more complex behaviour could be achieved without coding.

In this section, we discuss an evaluation of AGL. Our aim is to show that AGL is both essentially expressive and practically usable. We consider AGL as a type of specification language and adapt the DCSL evaluation approach that we applied in [4]. More specifically, we adapt from [28] the following three criteria for evaluating AGL: expressiveness, required coding level, and constructibility. We will present our evaluation of these criteria in Sections 8.1–8.3.

8.1. Expressiveness

This is the extent to which a language is able to express the properties of interest of its domain [28]. We measure the expressiveness of AGL from both structural and behavioral aspects. For structural aspects, the domain properties are captured as meta-concepts and associations in the language's ASM. For behavioral

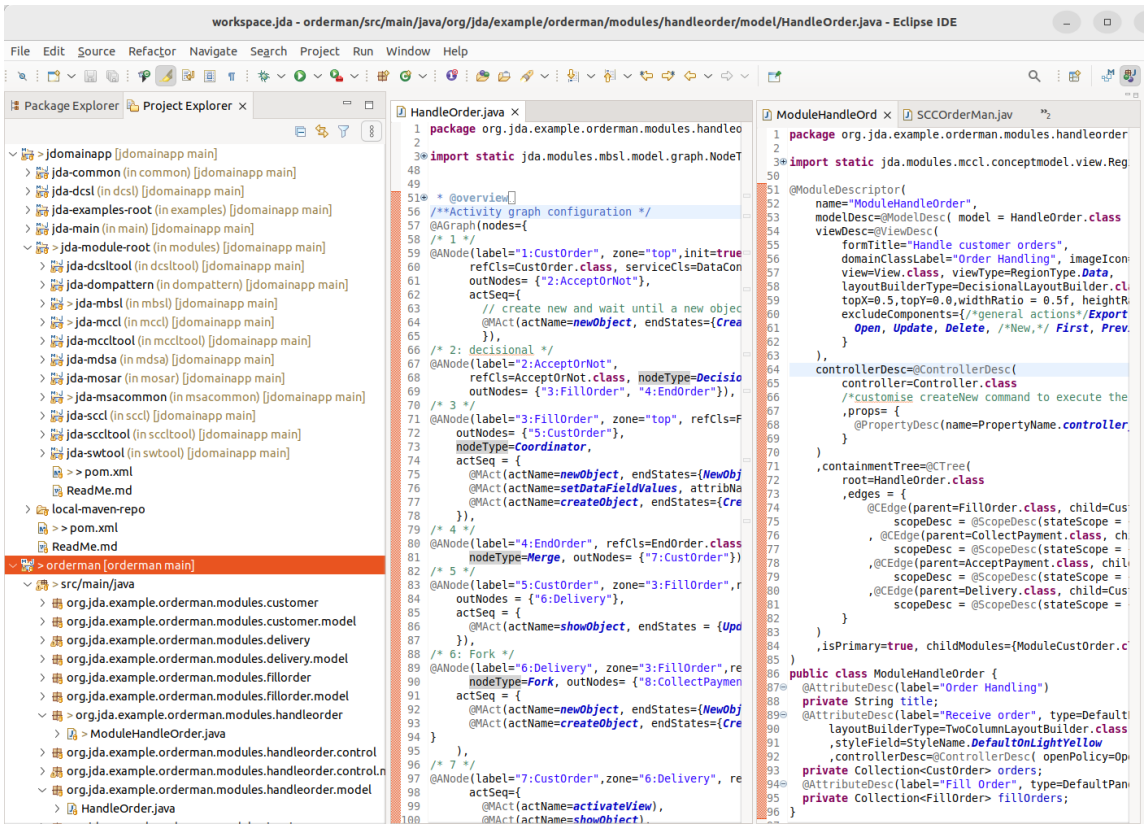


Figure 16: Illustration for the JDomainApp-based realization and usability of AGL.

aspects, AGL is able to express the five essential UML activity modeling patterns, as explained in Section 5. Therefore, any domain behavior captured by an activity diagram with these basic constructs could be expressed in AGL.

8.2. Required Coding Level

Reviewer: Third, compactness is evaluated by comparing the concrete and abstract syntax representations of the DSL. This is fine, although it would have been more convincing to compare with other approaches (not necessarily based on aDSLs).

Required coding level (RCL) complements the expressiveness criterion in that it measures the extent to which a language allows "...the properties of interest to be expressed without too much hard coding" [28]. Since AGL, to our knowledge, is the first aDSL of its type, we cannot compare AGL's RCL to other languages. Thus, we measure the AGL's RCL using the "compactness" of the language's CSM (see Sub-Section **??**). This is determined based on the reduction in the number of features in the CSM through the transformation ASM $\rightarrow$ CSM$_T$. More precisely, AGL's RCL is the percentage of the number of CSM$_T$'s features over the number of ASM's. The smaller this percentage, the higher the reduction in the number of features in the CSM and, thus, the more compact the CSM.

It is clear from Figures 11 and **??**(A) that AGL's RCL $= \frac{3}{9}$ or approximately 33%. Specifically, Figure 11 shows that the number of meta-concepts of the ASM involved in the transformation is 9. These exclude the four meta-concepts (`ActName`, `State`, `Decision` and `Join`) that are transferred directly to CSM$_T$. On the other hand, Figure **??**(A) shows that three meta-concepts result from the transformation (including `AGraph`, `ANode`, and `MAct`). Therefore, AGL can have a CSM that significantly reduces the number of meta-concepts required to write an AGC to only about one-third.

8.3. Constructibility

This is the extent to which a language provides "... facilities for building complex specifications in a piecewise, incremental way"[28]. 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 AGC at compile time. In addition, she can easily import and reference a domain class in an AGC and have this AGC automatically updated (through refactoring) when the domain class is renamed or relocated.

More importantly, the AGC can be constructed incrementally with the domain model. This is due to a property of our activity graph model (discussed in Section **??**) that the nodes and edges of an activity graph are mapped to the domain classes and their associations.

Further, we would develop automated techniques to ease the construction of AGC. Intuitively, for example, a technique would be to generate a default AGC for an activity and to allow the developer to customize it. We plan to investigate techniques such as this as part of future work.

8.4. Discussion and Limitations

Integration into a software development process is essential for the dissemination of our method in practice. We argue that our method is particularly suited for integration into iterative [29] and agile [30] development processes. In particular, the development team (which includes domain experts and developers) would use our tool to work together on developing the configured unified model in an incremental fashion: the developers use DCSL and AGL to create/update the configured unified model and then generate the software from this model. The domain experts give feedback for the model via the software GUI and the update cycle continues. The generated software prototypes can be used as the intermediate releases for the final software.

Further, in both processes, tools and techniques from **model-driven software engineering** (**MDSE**) would be applied to enhance productivity and tackle platform variability. In particular, we would apply PIM-to-PSM model transformation [31, 32] to automatically generate our configured unified model from a high-level one that is constructed using a combination of UML Class and Activity diagrams.

The usability of the software GUI, from the domain expert's viewpoint, plays a role in the usability of our method. Although in this paper we did not discuss this issue, we would argue in favor of two aspects of the software GUI, namely simplicity and consistency, which contribute towards its learnability [33]. Our plan is to fully evaluate GUI usability in future work. First, the GUI design is simple because, as discussed in [4], it directly reflects the domain class structure. Clearly, this is the most basic representation of the domain model. Second, the GUI is consistent in its presentation of the module view and the handling of the user actions performed on it. Consistent presentation is due to the application of the reflective layout to the views of all modules. Consistent handling is due to the fact that a common set of module actions (see Section 4) are made available on the module view.

Reviewer: I would suggest adding one paragraph discussing the generality of AGL. What would be the effort to implement the approach in other host languages? What features must a host language have to allow implementing DSCL/AGL on top of it?

9. Threats to Validity

This section discusses threats to validity of both our proposed method, the evaluation method. We organize threats according to the following four categories of validity in [34]: construct validity, internal validity, external validity and reliability.

9.1. Evaluation Method

The composition of the configured unified model in terms of the unified 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 the UML class and activity modeling languages that we currently use to construct the configured unified model. A question then arises as to the adaptability of our method to other behavioral modeling languages (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 unsatisfactory. 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 combined the class model with a behavioral model (e.g. a UML Activity diagram) into domain models constructing a configured unified domain model within a domain-driven architecture.

9.3. Internal validity

A concern with the internal validity of our case study is whether the CourseMan requirements sufficiently cover the activity graph that were discussed in Section 11. We incorporate in our definition of metamodel ASM for the abstract syntax of AGL and create a table Node objects, Edge objects, ModuleAct objects of the activity graph. We translate a behavior specification in the UML Activity diagram into a corresponding specification defined as a combination of pattern solutions (Domain behavior patterns) with an AGC provide transparent support for these. Our view is that although these are not the only design patterns in the five essential UML activity modeling patterns, our pattern-based approach could support domain behaviors that are specified by a UML activity with basic constructs corresponding to these patterns.

9.4. External validity

Threats to external validity of our method include those that impact how our method is applicable to the development of other MSA-based and DDD-based software that have similar characteristics. The first threat stems from a fact that our method is applicable to systems that are designed based on MDSA (a combination of MSA and DDD). The second threat is the generality of the case study. One would argue whether or not the case study that we selected is representative of the real-world ones. we approach the modeling patterns helps mitigate this threat because it is based on two well-known software design principles to keep the patterns generic, for each pattern form a UML activity model and a template configured unified model that realizes it has similar characteristics would be handled in the same way.

10. Related Work

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

DSL Engineering. DSLs [35, 36] can be classified based on the domain [6], as vertical or horizontal, or based on the relationship with a host language [18, 35, 36], 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. To the best of our knowledge, AGL is the first aDSL that is 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 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 interaction diagrams would be used to model this behavior.

In UML [5, p. 285], Interaction diagrams (such as Sequence diagrams) are only one of three main diagram types that are used to model the system behavior. The other two types are State machine and Activity diagram. Although in the book, Evans only uses sequence diagrams as an example, in the ApacheIsis framework [2] 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 behavior 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 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. There have been works attempting to combine UML structural and behavioral diagrams to construct a system model, similar in spirit to the unified model that we proposed in this paper. Intuitively, this makes sense because the two diagram types address the two core (static and dynamic) aspects of a system. Two works [37, 38] discuss combining UML class and state machine diagrams to model the system. Another work [39] explains the relationships between UML structural and behavioural diagrams and how these relationships can be leveraged to build a complete system model. In particular, this work highlights a strong relationship between state machine (a.k.a statechart) and activity diagram – an insight that we also discovered in this paper.

Our proposed unified domain modeling is novel in that it combines UML Class and Activity diagrams by incorporating the domain-specific structure (activity class and associations) into the class diagram, thereby creating a unified model. In the spirit of the DDD's layered architecture, we separated the activity graph component of Activity diagram from the unified model and created a separate aDSL (AGL) for it. The unified 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, 32]. 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 [40, 41] 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 works use four DSLs and do not clearly indicate which ones are used for constructing the domain model and which are used to build other parts of the software model.

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 configured unified model, and (2) how this model is used to automatically generate the entire software.

11. Conclusion

In this paper, we proposed a unified modeling method for developing object-oriented domain-driven software. Our method consists in constructing a configured unified domain model in the MOSA architecture. The unified model is an extension of the conventional domain model to incorporates the domain-specific features of the UML Activity diagram. It is expressed in DCSL, which is an aDSL that we developed in previous work. To use the unified model at the core layer of MOSA, we developed another aDSL named AGL to express the domain behaviors for a unified model. We used the annotation attachment feature of the host OOPL to attach an AGL's activity graph directly to the activity class of the unified model, thereby creating a configured unified model. We systematically developed a compact annotation-based syntax of 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 evaluated AGL to show that it is essentially expressive and practically suitable for designing real-world software.

We argue that our method significantly extends the state-of-the-art in DDD on two important fronts: bridging the gaps between model and code and constructing a unified domain model. Our proposed aDSLs are horizontal DSLs that can be used to support different real-world software domains. Our plan for future work includes developing an Eclipse plug-in for the method and developing graphical visual syntaxes for DCSL and AGL.

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. doi:10.1016/j.infsof.2019.106239.
[9] G. Booch, Object-Oriented Development, IEEE Transactions on Software Engineering SE-12 (2) (1986) 211–221. doi:10.1109/TSE.1986.6312937.
[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, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 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] A. v. Lamsweerde, Formal Specification: A Roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 147–159.
[29] 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.
[30] 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.
[31] 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.
[32] M. Brambilla, J. Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice, 1st Edition, Morgan & Claypool Publishers, 2012.
[33] E. Folmer, J. Bosch, Architecting for Usability: A Survey, Journal of Systems and Software 70 (1–2) (2004) 61–78. doi:10.1016/S0164-1212(02)00159-0.
[34] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.
[35] A. van Deursen, P. Klint, J. Visser, Domain-specific Languages: An Annotated Bibliography, SIGPLAN Not. 35 (6) (2000) 26–36.
[36] 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.
[37] 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.
[38] 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.
[39] P. Selonen, K. Koskimies, M. Sakkinen, Transformations Between UML Diagrams, JDM 14 (3) (2003) 37–55. doi:10.4018/jdm.2003070103.
[40] J. Warmer, A Model Driven Software Factory Using Domain Specific Languages, in: Model Driven Architecture- Foundations and Applications, Springer, 2007, pp. 194–203.
[41] J. Warmer, A. Kleppe, Building a Flexible Software Factory Using Partial Domain Specific Models (Oct. 2006).