AGL: Incorporating Behavioral Aspects into Domain-Driven Design

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

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

Abstract

Domain-driven design (DDD) aims to iteratively develop software around a realistic domain model, i.e., that can be viewed as a program from the developer's perspective. 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 diagrams and statecharts, are not explicitly captured as part of the domain model. This approach requires an additional and also challenging effort to integrate them into the domain model for a realistic one. In this paper, we propose a novel unified domain modeling method to tackle this. We employ our previously-developed aDSL, named DCSL, in order to express our unified domain model, that includes new domain classes extracted from the underlying behavioral models within the context of a module-based software architecture (MOSA). We then define a novel aDSL, named activity graph language (AGL), to capture the remaining behavioral aspects. The mapping between the two aDSL models allows us to obtain a unified and realistic domain model as an incoporation of the behavioral models and the original domain model. To realize our method, we define a compact annotation-based syntax meta-model and then a formal semantics for AGL. We demonstrate our method with an implementation in a Java framework named jDomainApp and evaluate AGL to show that it is essentially expressive and usable for real-world software.

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. Recent works in DDD [2, 3] have been focusing on using annotation-based domain-specific languages (aDSLs) to build the domain model. An aDSL is internal to a host object-oriented programming language (OOPL) and consists in a set of annotations specifically designed to express the domain concepts. The resulted domain model is used as the base input for a code generator to generate the entire software. Within these works behavioral aspects, that are often represented using UML activity diagrams and statecharts, are not explicitly captured as part of the domain model. They are either implicitly embedded into the code generator or incorporated at the code level.

In this paper, we propose a novel unified domain modeling method to tackle these. We view unified model as an extended domain model in a module-based software architecture (MOSA) [4–6] that we have recently developed for DDD. Our method both complements and extends another previous work [7], in which we use UML activity diagram [8] to model software behavior and introduce the notion of a unified

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

domain model. We choose UML activity diagram for behavioral modeling because this language has been shown to be domain-expert-friendly [9] and that it can be used to design at different layers of abstraction. However, we define in [7] only an abstract unified model and take for granted the UML activity graph that helps integrate this model into the architecture.

Our approach in this paper is to first provide a precise definition of unified model using a horizontal aDSL named DCSL [7]. We consider horizontal aDSL (a.k.a technical aDSL) as an aDSL that targets a technical (low-level) domain, whose concepts describe design patterns that often underlie a class of real-world (high-level) domains. We then propose a novel horizontal aDSL, named activity graph language (AGL), for expressing the UML activity graph. AGL's domain is a restricted domain of the UML's activity graph language [8]. The restriction is necessary only to incorporate this language into MOSA. We assert that executable nodes of the activity graph execute actions on the software modules in MOSA. To enable this, we define a set of essential module actions which concern the manipulation of the domain objects of the domain class that is owned by each software module.

A key benefit from combining DCSL and AGL in MOSA is that it helps define a complete executable model for the software. Further, this software is automatically generated using a Java software framework, named jDomainApp [10], that we have developed.

As far as language specification is concerned, we adopt the meta-modeling approach for DSLs [11] and use UML/OCL [8, 12] to specify the abstract and concrete syntax models of AGL. In particular, we propose a compact annotation-based concrete syntax model that includes a very few number of concepts. We systematically develop this syntax using a transformation from a conceptual model of the activity graph domain. 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:

- precisely define unified model in the context of MOSA
- define a set of essential module actions for the software modules of MOSA
- specify an aDSL (named AGL) for constructing the activity graph. AGL has a compact syntax.
- demonstrate our method with an implementation in the jDomainApp framework
- evaluate 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 a technical background. Section 3 explains our approach to incorporating behavioral aspects into a domain model. Section 4 defines the essential module actions. Section 5 specifies AGL. Section 6 discusses tool support. Section 7 presents the AGL's evaluation. Section 8 discusses threats to validity of our work. Section 9 reviews the related work and Section 10 concludes the paper.

2. Motivating Example and Background

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

2.1. Motivating Example

We focus on the course management domain (CourseMan) as our motivating example.

Figure 1 on the left-hand side presents a UML/OCL class diagram that represents the **domain model** of CourseMan. Class `Student` represents the domain concept Student[1]. Class `Course` represents the Courses[2] that are offered to students. Class `SClass` represents types of classes (e.g. morning, afternoon, and evening) that `Student`s can choose.

[1] use `fixed font` for model elements and normal font for concepts.
[2] use class/concept name as countable noun to identify instances.

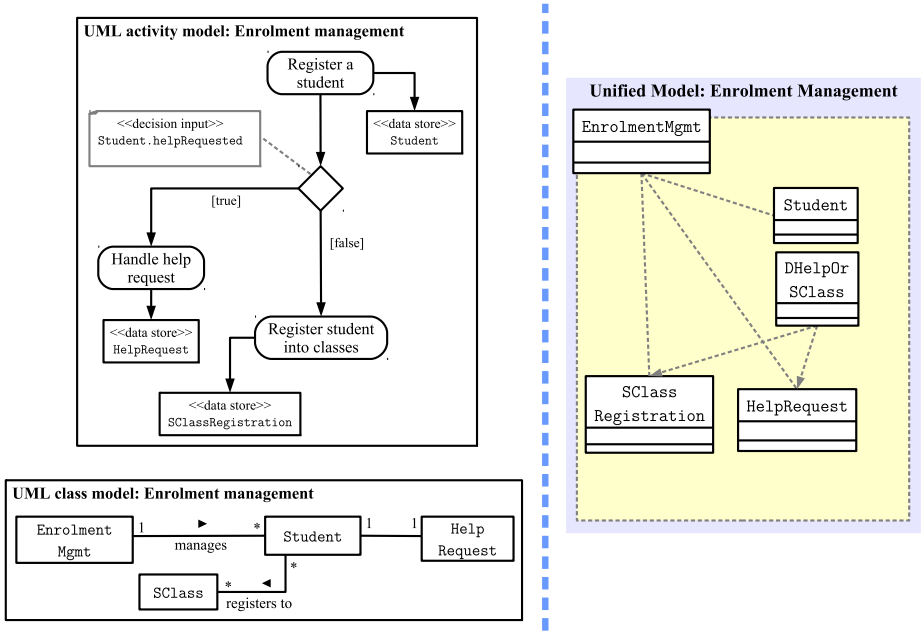

Figure 1: Simplified UML/OCL class and activity diagrams to specify a CourseMan software variant that handles the enrolment management activity.

The right-hand side of Fig. 1 depicts an activity diagram that captures behavioral aspects from the enrolment management activity.
TODO:
- add OCL
- add the activity diagram EnrolmentMgmt.

2.2. Domain Models in the Annotation-Based Domain Specific Language DCSL

Annotation-Based Domain Specific Language (**aDSL**) is coined in [13] as an attempt to formalise the notion of fragmentary, internal DSL [14] for the use of annotation to define DSLs. An aDSL is defined based on an OOPL's abstract syntax model [7] 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 [15] and C# [16].

We stated in [7] 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.

Within the scope of this paper and based on the DSL classification in [11], we differentiate between two types of aDSL: horizontal and vertical aDSLs. A *vertical aDSL* targets a bounded real-world (vertical) domain. In contrast, a *horizontal aDSL* (a.k.a technical aDSL) targets a technical (low-level) domain, whose concepts describe the patterns that often underlie a class of vertical domains that share common features. More specifically, a horizontal aDSL is a DSL internal to a host OOPL, whose domain is a technical one and that uses a set of annotations to model the domain concepts.

Domain class specification language (**DCSL**) [7] 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 realises 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` with a certain behavior type. 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.

Syntactically, we write a DCSL model directly using the host OOPL's syntax. For exposition purposes, however, we write this model using an extended UML graphical notation that uses a *structured text box* for writing annotations. Specifically, nonannotation elements are drawn using the usual UML class diagram notation. On the other hand, the annotation elements are drawn using UML note box. Annotation assignment is represented by a dashed grey line, whose target element end is marked with the attachment symbol (■). The note box content has the form A {*props*}, where A is the annotation name and *props* is a property listing. Each entry specifies the initialisation of a property to a value. If this value is another annotation element then this element is written using a nested, borderless note box. The entries are separated by either a next line or a comma (',') character.

Another feature of the above notation is the use of a virtual (dashed) association line to represent a pair of `DAssoc` elements that help realise the association ends of an association. This association line is more compact and thus helps significantly ease drawing and improves readability of the model. We will often use the term "association" to refer this association line and the DCSL model elements that realise it.

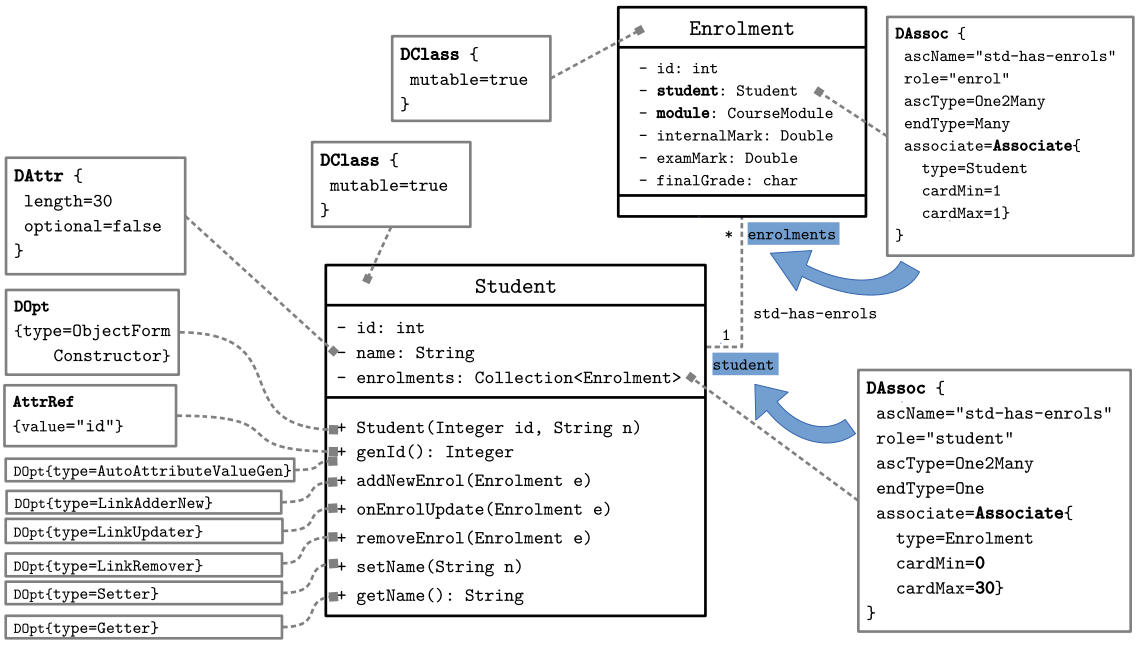

Figure 2: A partial CourseMan domain model expressed in DCSL (adapted from [7]).

For example, Fig. 2 shows a partial CourseMan's domain model expressed in DCSL. This model involves two domain classes: `Student` and `Enrolment`. Class `Enrolment` realises the many-many association between `Student` and `Course` (encapsulated by `CourseModule` as explained in SubSect 2.3). Both `Student` and `Enrolment` are assigned with a `DClass` element, which state 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 domain field `Student.id`. This means that `genId` is the method that automatically generates values for `Student.id`.

2.3. The Module-Based Software Architecture MOSA

To construct DDD software from the domain model requires an architectural model that conforms to the generic layered architecture [1, 17]. 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 [18] is one such model. The existing DDD frameworks [2, 3] supports 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 backbone of any DDD tool that conforms to the DDD's layered architecture.

Our previous works [4–6] propose 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 [8] 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 parameterised 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 [5, 6] 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 [10] to automatically generate a software. In this paper, we will assume that a module class is defined for every domain class.

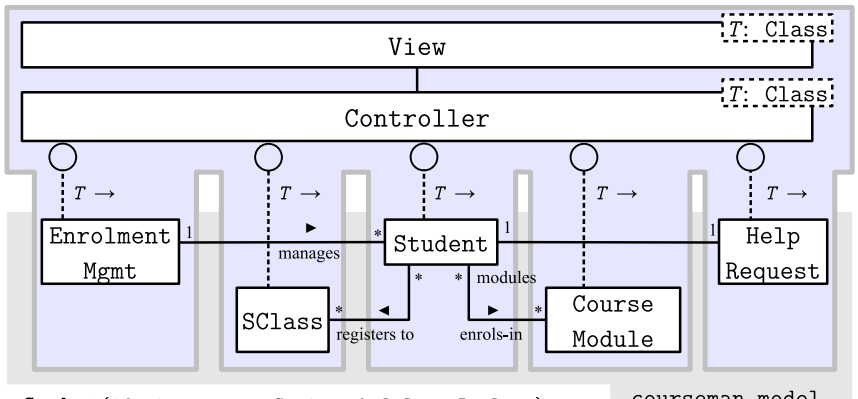

Figure 3: The MOSA model of CourseMan.

To illustrate, the top-half of the MOSA model in Figure 3 shows five module classes of 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 is `Student`, the view is `View⟨Student⟩` and the controller is `Controller⟨Student⟩`.

We argue that MOSA captures the essence of object-oriented software design in a modular, MVC-based design structure. According to Booch [19], an object-oriented software consists in objects and their interactions that are realised though behavior invocation. Given that the domain model is expressed in DCSL, 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 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 [10], 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.

However, in [7] we scoped our use of MOSA at the boundary of the domain model and assumed that this model is connected to the rest of MOSA model via activity graph. To express this graph in the context of MOSA requires exposing the component interface of the software modules and connecting this interface to the graph. We call this interface the module interface and discuss its design in Section 4. We propose a language for expressing activity graphs in Section 5.

2.4. Motivating Questions

- What are behavioral apspects?
- Why MOSA?
- What are challenges? (or why agl?)

3. Incorporating Behavioral Aspects into a Domain Model

This section ...

3.1. Basic Idea

Modeling a software in DDD requires a domain model that supports both structural and behavioral aspects. In our method, we express these using a combination of **unified domain model** and **activity graph**. Figure 4 shows our proposed method, which has been refined from the method that we introduced in [7]. The figure highlights not only the unified model's role but its combination with an activity graph. We consider the unified model as an extended domain model in MOSA. This model, which is expressed in DCSL, extends the conventional DDD's domain model [1] with the domain-specific features of UML activity diagram. Among the essential features that are supported include activity class (e.g. class $\mathtt{C_a}$ in Figure 4), which represents an activity. We use the activity class of each unified model as a pivot with which to define the activity graph. Each activity class is attached to an activity graph that describes the behavioral logic of the represented activity. The activity graphs are expressed in AGL.

Hence, conceptually our method consists in iteratively performing three steps. The first step takes as input the domain requirements, optionally expressed in some high-level models (e.g. UML class and activity diagrams), and creates a set of initial unified domain models and associated activity graphs. At this stage, the models and their graphs may be incomplete and, thus, need to be refined in subsequent iterations. The second step takes as input the unified models and graphs and uses MOSA to automatically generate a GUI- and module-based software. This software is presented to the domain expert to get feedbacks. If there are feedbacks then the third step updates the unified models and graphs and the cycle continues. If, on the other hand, the domain expert is satisfied with the models and graphs then the cycle ends.

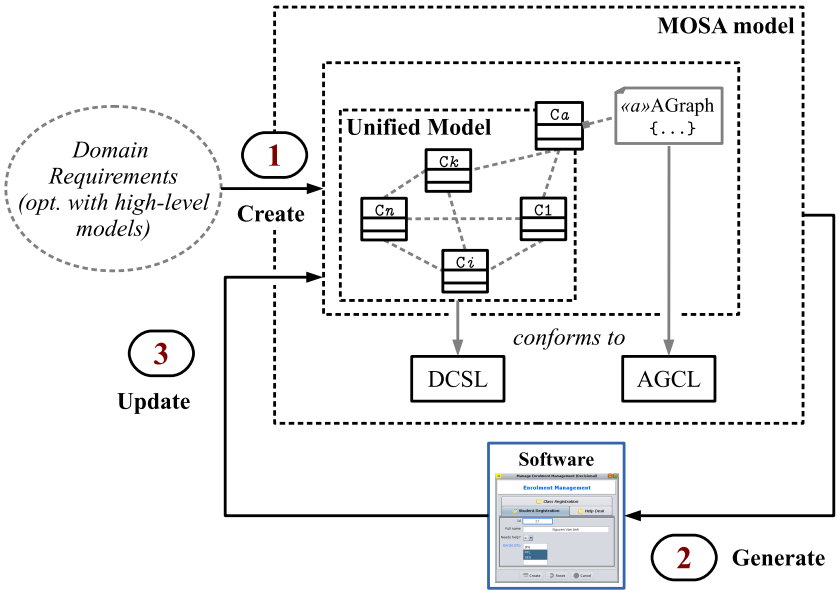

Figure 4: An overview of our method.

3.2. "UDM Specification"

In principle, unified model is a DCSL model that realises what we term the UML *unified class model*. This model extends the conventional domain model [1] with a domain-specific structure from the activity modeling domain.

Definition 1. *A **unified class 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 (resp. 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 following class pairs:*
  - *activity class and a merge class.*
  - *activity class and a fork class.*
  - *a merge (resp. fork) class and a data class that represents the data store of an action node connected to the merge (resp. 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. 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 realises an unified class model as follows:*

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

Example: Unified model

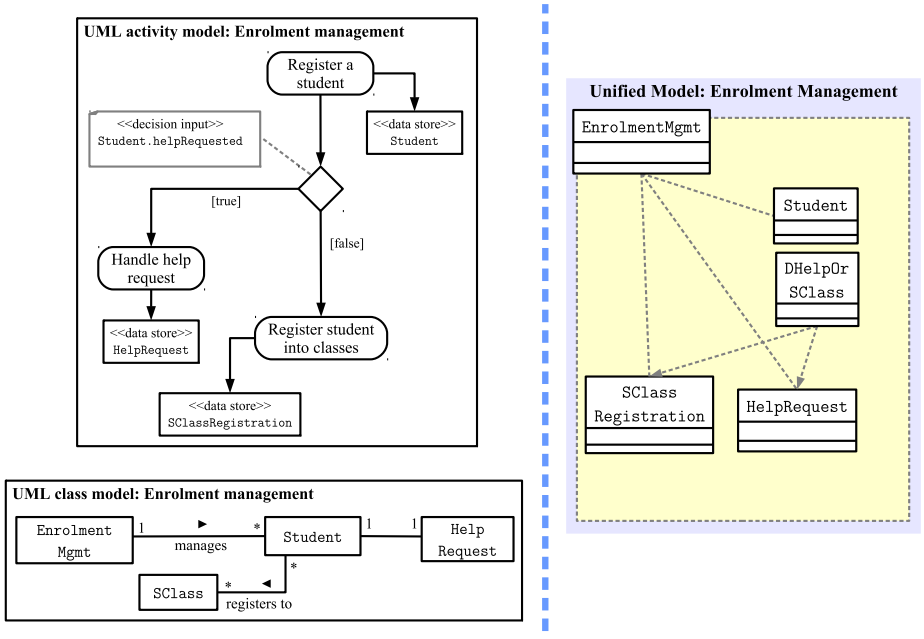

Figure 5: (A: Left) The UML activity and class models of a CourseMan software variant that handles the enrolment management activity; (B: Right) The unified model that results.

To illustrate, Figure 5(A) shows the UML activity and class models of a CourseMan variant that handles the enrolment 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 shown in Figure 3. Figure 5(B) shows the resulted unified model of the activity. This model consists

of five domain classes and realisations of five activity-specific associations. To ease reading, we omit the domain-specific associations that are shown in the UML class model in Figure 5(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 realise 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 realise the data stores connected to the two actions 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.

In Section 7.1, we will revisit this example in the context of the decisional modeling pattern and present a software GUI that is generated from the model.

4. Domain Behaviour Semantics

Using the unified model at the core of a MOSA model requires defining for the modules a set of essential actions for manipulating the domain objects of a domain class. We consider these actions as forming a module interface, which is represented by a UML interface named `ModuleService`. In this section, we provide a formal definition of module action, which is suitable for use with the activity graph language that we propose in Section 5. Our definition focuses on describing the structure of module action and its pre- and post-states. We base our formalism on the UML Action language (§16[3] [8]), which incorporates the notion of state. State is argued to be an intrinsic part of behavioral specification (§14 [8]). 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*.

4.1. Expressing Domain Behaviour with UML Activity Diagram

behavior -> activity
behavior structure -> control and action nodes
We focus on providing the semantics for Action.

4.2. Module-Based Action Semantics

4.2.1. Atomic Action

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

Definition 3. *An **atomic action** is a smallest meaningful module behavior provided to a user (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` *[8]): the states at which a current module must be in in order for this action to proceed.*
- `postStates` *(for* `localPostcondition` *[8]): the states at which the action completes its execution on a current module.*

[3]We use '§' to denote both Chapter and Section, which is interpreted from the context.

- `fieldValSet` *(for* `input` *[8]): captures the input of the action. It is a set of pairs* (f, v) *where* f *is name of a domain field of the domain class and* v *is the value that is to be set 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 followings 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 4.2.4) 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: 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).

Table 1 lists definitions of the core atomic actions. For exposition purpose, 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 include three essential domain object manipulation actions: `createObject`, `updateObject` and `deleteObject`. The post-states of these actions include the following states: `Created`, `Updated` and `Deleted` (*resp.*).

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

4.2.2. 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 (§16.11 [8]). Denote by `first` and `last` two functions that return the first and last elements (*resp.*) in a sequence.

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

S has the following properties:

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

□

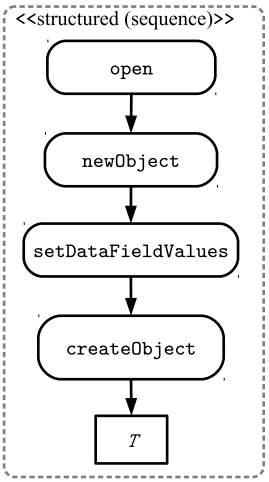

Figure 6: An ASE that creates a new domain object of a module's domain class (typed T) .

For example, Figure 6 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`", `postStates` = { `Created` } and
`fieldValSet` = `setDataFieldValues.fieldValSet` = $\emptyset$.

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 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 behaviours 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 6) 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.

4.2.3. Reachable States

The definition of ASE gives rise to a 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'.* □

Table 2: The core atomic actions and their reachable states

Actions	Reachable states
`open`	`Opened`, `NewObject`, `Editing`, `Created`, `Updated`, `Deleted`, `Reset`, `Cancelled`
`newObject`	`NewObject`, `Editing`, `Created`, `Reset`, `Cancelled`
`setDataFieldValues`	`Editing`, `Created`, `Updated`, `Reset`
`createObject`	`Created`
`updateObject`	`Updated`
`deleteObject`	`Deleted`
`reset`	`Reset`
`cancel`	`Cancelled`

Clearly, the post-state of an atomic action is reachable from its own action. Table 2 lists the reachable states of every atomic action defined in Table 1. The first row shows how action `open` can reach all other states. 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 state `Opened`, because this state is raised only once. The second row shows how 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. The third row shows how action `setDataFieldValues` cannot reach `NewObject`, `Deleted` and `Cancelled`. This is because this action concerns only with inputting data and thus cannot initiate or cancel object creation, nor can it lead to object deletion. The last five rows of the table show that the corresponding five actions each has only one reachable state, which are their own states. These actions are "stubs", in the sense that they terminate all the ASEs that lead to them.

For example, the ASE in Figure 6 shows that state `Created` is reachable from any of the three member actions that precedes the action `createObject`. These include `open`, `newObject` and `setDataFieldValues`.

4.2.4. 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 $A = \{S \mid \texttt{first}(S) = a, S.\texttt{postStates} \subseteq E\}$, 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 $i = \emptyset$ then we omit it and simply write A as $(a, \{s_1, \ldots, s_n\})$. □

Clearly, SAA generalises both atomic action and ASE: an ASE is a single-member SAA, while an atomic action a is the SAA $(a, \{a.\texttt{postState}\}, a.\texttt{fieldValSet})$. 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.

For example, let us consider the SAA $(\texttt{newObject}, \{\texttt{Created}, \texttt{Cancelled}\})$, which represents a common ASE set that starts with 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 6 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. Module-Based Domain Behavior Language

The unified model is linked to an activity graph, which models the generic graph structure that is common to all activities. This activity graph incorporates module action to specialise the behavior of its nodes. In the terminology of the DDD's layer architecture [1], the activity graph is positioned at the application layer, because it coordinates the behaviors of the modules owning the domain classes in the unified model in order to perform the overall activity's behavior.

From the language engineering's perspective, we argue that the same benefits that are gained in unified domain modeling with DCSL can be attained for activity graphs if we develop a horizontal aDSL for them. We call this aDSL **activity graph language** (**AGL**). 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 [11], we specify AGL in terms of an *abstract syntax meta-model* (ASM) and an annotation-based textual *concrete syntax meta-model (CSM)*. We also briefly discuss the semantics of AGL, relative to the activity graph and module action. More specifically, we construct the ASM in three steps. In the first step, we construct a conceptual model (CM) of the domain as a UML/OCL class diagram. This model helps understand the overall structure, without being constrained by the target OOPL's meta-model. The next two steps gradually transform CM into the ASM. In the second step, we transform CM into an equivalent, annotation-friendly form, called CM_T. In order to reduce the size of the eventual ASM, we try, in this step, to produce a compact CM_T. In the third step, we then transform CM_T into the actual ASM, which takes an annotation-based form specified by the OOPL's meta-model. In this form, the configuration-related meta-concepts are represented by annotations. Further, we use Class as the basis to structure the annotations.

5.1. Domain

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 stated in Chapters 15 and 16 of the UML specification [8]:

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 (§15.2.3.3, pg. 374) is only applied to decision node.

X1. using variable with activity (§15.2.3.5, pg. 417).

X2. variable action (§16.9, pg. 467).

X3. activity edge (§15.2.3.3, pg. 373) is without guards.

I1 and R1 are needed to incorporate activity graph into MOSA. R2 is a safe restriction because, according to the specification, value specification is mainly used for specifying conditions on decision node. X1 and X2 concern the use of variables. According to the UML specification, variable is alternative to using object flow. The exclusion of edge guards in X3 is not a limitation of our approach. It is a deliberate omission at this stage when we want to focus on supporting the core structure of the activity graph. We plan to remove X3 in future work.

5.2. Conceptual Model (CM)

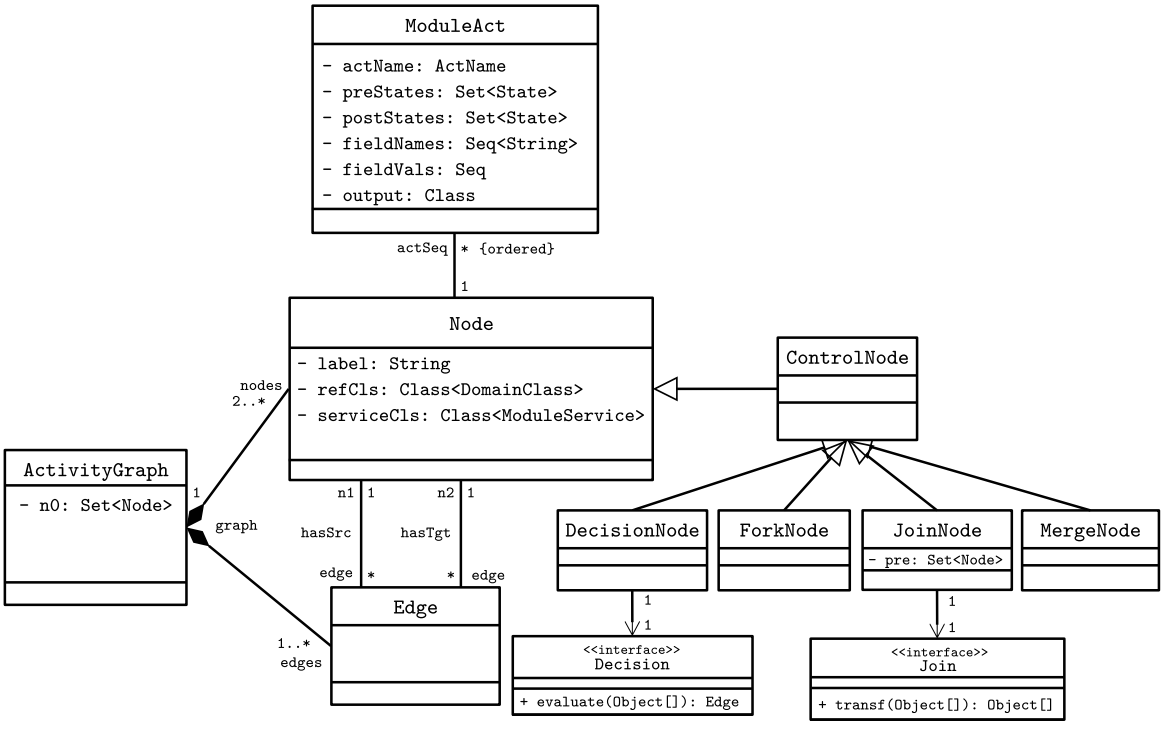

Figure 7: The conceptual model (CM) of AGL.

We define the CM using the UML class model shown in Figure 7. The well-formedness OCL rules of this model are presented in Appendix A. To unify the notation with the unified model, in the text we will express

the concepts of this model using the equivalent DCSL's terms (see Section **??**). This is possible because the model only contains elements (class, attribute, one-one and one-many associations and generalisation) that are expressible by DCSL. The following paragraphs describe the main meta-concepts of the CM. 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 4.2.1).

`ModuleAct`

This represents SAA-typed module actions as defined in Definition 6. Field `actName` realises the action name. The three fields `preStates`, `postStates` and `output` realise three similarly-named attributes of the action. The two fields `ModuleAct.fieldNames` and `fieldVals` together realise the attribute `fieldValSet` of the action, as follows: each pair (f, v) in `fieldValSet` is constructed by taking f from `fieldNames` and v from the corresponding element of `fieldVals`.

`ActivityGraph`

This represents activity graphs and has three fields: `nodes`, `edges`, and `n0`. The first two fields are associative fields that realise the associations to `Node` and `Edge` (*resp.*). Field `n0` realises a sub-set of nodes that are the start nodes of the graph. The start nodes are the ones that are executed first when the graph is executed.

`Node`

This represents activity nodes and has four fields. Field `label` realises the node label. The next two fields specify the **referenced (abbrv. *ref*) software module**, i.e. the module that is referenced by this node. Specifically, field `refCls` (typed `Class⟨DomainClass⟩`) specifies the domain class of the *ref* module. We call this class the *ref* domain class. Here, we assume `Class⟨DomainClass⟩` represents the Domain Class concept of DCSL (see Section 2.2).

Field `serviceCls` (typed `Class⟨ModuleService⟩`) specifies the actual `ModuleService` class of the *ref* software module. A default module service class for action nodes that we developed as part of the JDomainApp framework [10] is a class named `DataController`. It is through a module service object of `serviceCls` that the current `Node` is able to perform the `ModuleAct`s specified by the field `actSeq`. This field is an associative field that realises the the association from `Node` to `ModuleAct`.

`ControlNode`

This is an abstract sub-type of `Node` that represents the control nodes of the activity graph. This class is used to specify behavior of control nodes and to capture the state of its execution. We specialise class `ControlNode` into the four sub-types: `DecisionNode`, `ForkNode`, `JoinNode`, and `MergeNode`. In particular, class `DecisionNode` references an interface named `Decision`, which provides a method (named `evaluate`) for evaluating the decision logic. Similarly, class `JoinNode` references interface `Join`, which has a method (named `transf`) for transforming the input tokens into output ones (if needed). Further, class `JoinNode` has a field named `pre`, which is a derived field that realises the source `Node`s of the activity edges connecting to a `JoinNode`.

Actual implementations of the interfaces `Decision` and `Join` are provided in the corresponding decision and join classes (*resp.*) in the unified model.

`Edge`

This represents activity edges. It has two associative fields `n1` and `n2`, which realise the two associations to `Node`. Field `n1` captures the source node, while field `n2` captures the target one. Intuitively, there is a correspondence between an `Edge` and an association between the two domain classes that are referenced by the source and target nodes of the edge.

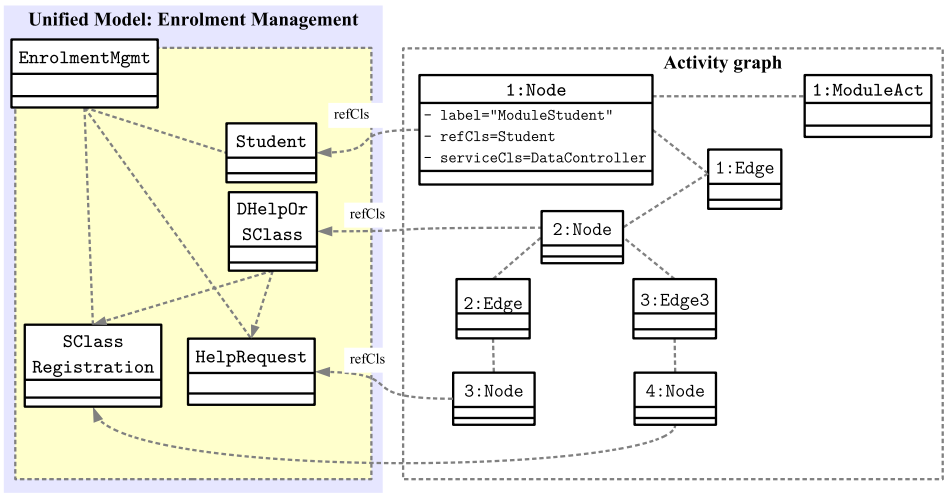

Figure 8: (LHS) A repeat of the unified model shown in Figure 5; (RHS) The activity graph of this model.

Table 3: (A: Top) `Node` objects, (B: Bottom-left) `Edge` objects of the activity graph in Figure 8 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: Activity graph

The right hand side of Figure 8 is an activity graph of the enrolment management activity of the CourseMan software variant introduced earlier in Section **??**. The left hand side of the figure is the corresponding unified model of this activity, which is repeated from Figure 5 to show links with the activity graph. We will discuss other example graphs that include control nodes later in Section 7.1. Tables 3(A) and (B) list the states of the nodes and edges (*resp.*) of the activity graph. Table 3(C) lists the `ModuleAct` objects that are referenced by the `Node`s in Table 3(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`. It also references object 1:`ModuleAct`. The `refCls`'s value of each node is depicted in the figure by a dashed curve (labelled "`refCls`") that connects the node to the referenced domain class in the unified model.

5.3. Abstract Syntax Model (ASM)

Our main objective is to construct an ASM from the CM by transformation, so that the ASM takes the annotation-based form, suitable for being embedded into a host OOPL. Furthermore, we will strive for a compact ASM that uses a small set of annotations. From the practical standpoint, such a model is desirable since it will result in a compact concrete syntax, which requires less effort from the language user

to construct an MCC. To achieve this requires two steps. First, we transform CM into another model, called CM_T, that is compact and suitable for annotation-based representation. Second, we transform CM_T into the actual annotation-based ASM. We first explain CM_T and the transformation CM $\rightarrow$ CM_T. After that, we explain the ASM.

5.3.1. CM_T: A Compact and Annotation-Friendly Model

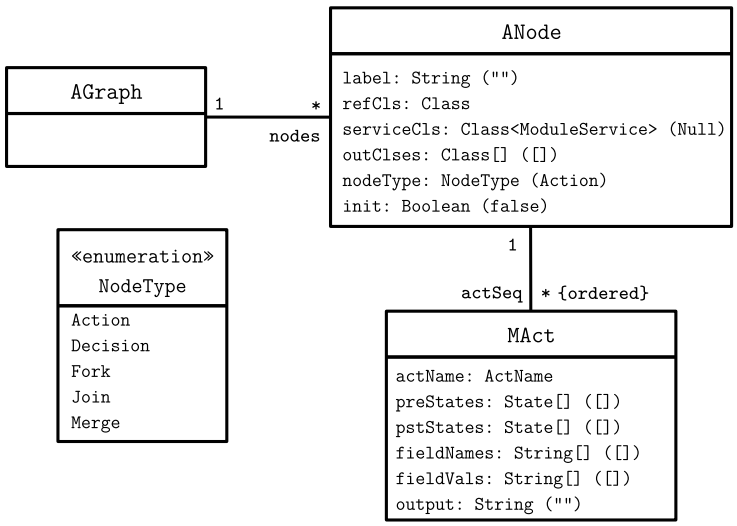

Figure 9: CM_T: a compact and annotation-friendly model.

Figure 9 shows an annotation-friendly version of the CM, called CM_T, which consists of three meta-concepts: activity graph (`AGraph`), activity node (`ANode`) and module action (`MAct`). To ease discussion later about the annotation-based ASM, we add to the figure the default value notation of the optional domain field (i.e. field with `DAttr.optional` = `true`). The default value is written within a pair of brackets that immediately follow the field's data type. We briefly describe below the three meta-concepts of CM_T. The precise meaning of these meta-concepts will be explained through a transformation that we define in the next section.

Note that due to the restrictions on the data type of an annotation property, fields of certain meta-concepts in the CM are not translated directly to fields in the CM_T. In these cases, however, we compensate for the information loss by adding OCL constraints to the corresponding meta-concepts of the CM_T. These constraints are realised by validation functions that are performed on these meta-concepts, when they are translated into the annotation form.

`MAct`

`MAct` realises `ModuleAct` using only the data types that are supported by annotation. Specifically, the data types of `MAct.preStates` and `pstStates` (the latter is short for `postStates`) are arrays of `State`. The default values of these fields are an empty array ([]), which do not mean that they are not specified. An empty array in this case means that it takes the default state value of action as specified in Table 1 of Section 4.2.1. The following additional OCL constraints help ensure that the two fields contain unique values, which are required to match the `Set` data type of the two corresponding fields of `ModuleAct`.

```
1 -- MAct.preStates and pstStates (if specified) contain unique values
2 context Node inv:
3   not(preStates.oclIsUndefined()) implies preStates->asSet() = preStates and
4   not(pstStates.oclIsUndefined()) implies pstStates->asSet() = pstStates
```

As for the two fields `MAct.fieldNames` and `fieldVals`, they also take an array type. This is equivalent to the `Seq` data type of the two corresponding fields of `ModuleAct`. Note that `fieldVals` is typed `String[]`, which requires that the value objects, if specified, are written explicitly as a string. Fortunately, this is not at all troublesome, because `fieldVals` is only required if the value objects are specified by the user. For many cases, however, the values come from another action or an external system. In these cases, `fieldVals` need not be specified and can take the default value of an empty array.

Last but not least, field `MAct.output` is typed `String` and has the default value of an empty string (“”). This field is added only for completeness. It always takes the default value, because the output value of a module action is never specified by the user. It is generated from within the system.

`ANode`

Class `ANode` both represents `Node` and `Edge` and merges the entire `ControlNode` type hierarchy. To achieve the former, we add to `ANode` a new field, named `outClses`, that captures the *ref* domain classes of the target nodes of the outgoing edges of a node. To achieve the latter, we add to `ANode` a field named `nodeType`, whose data type is the enumeration `NodeType`. This enumeration specifies all the pre-defined node types, including action and the control types.

```
1 -- ANode.refCls and ANode.outClses (if specified) are domain classes
2 context Node inv:
3   not(refCls.oclIsUndefined()) implies refCls.isDomainClass() and
4   not(outClses.oclIsUndefined()) implies outClses->forAll(isDomainClass())
```

Note that we cannot explicitly define the data types of `ANode.refCls` and `outClses` as parameterised types of `DomainClass`, because this class only exists in the CM and not in the actual annotation-based model. We compensate for this information loss in the two data types by two OCL constraints on `ANode` for the two fields. Both constraints (listed immediately above) make use of a boolean function named `isDomainClass`. This function, which is defined as part of the CM's library rules in Appendix A.6, is invoked on a class to check if it is attached to a `DClass` element

`AGraph`

Class `AGraph` is simplified from `ActivityGraph` by having just one associative field for `ANode`. To further simplify this graph and ease its configuration, we replace the field `ActivityGraph.n0` by a new boolean-typed field `ANode.init`. We reconstruct `ActivityGraph.n0` from all `ANode`s that have `init` = `true`.

5.3.2. Mapping from CM to CM$_T$

We explain in this section the precise transformation CM $\rightarrow$ CM$_T$ in terms of a set of mapping rules. Table 4 lists definitions of these rules. Mapping rule M_1 is an identity map on `Decision`, `Join`, `ActName`, and `State`. These meta-concepts are transferred directly to CM$_T$. Given the additional OCL constraints that were defined previously on `MAct`, mapping rule M_2 maps `ModuleAct` to `MAct`.

Mapping rules M_3–M_9 define the mapping for `ANode`. Specifically, M_3 maps `Node` to the field set of `ANode` that excludes (`excl`.) three fields (`nodeType`, `outClses`, and `init`). The other rules define mapping for these excluded fields. First, rules M_4–M_7 together map the four `ControlNode` subtypes to `ANode.nodeType` (this field specifies four subsets of `ANode` objects). Second, rule M_8 maps to field `ANode.outClses` a query function, named `outClses`, that returns a set of domain classes derived from the field `Node.refCls` of the target nodes (n_2) of all the outgoing edges (n.`out`) of a `Node` (n). Here, `Node.out: Set`$\langle$`Edge`$\rangle$ is a derived field, whose value consists of all `Edge`s that have field `n1` equating the current node. Third, rule M_9 maps to field `ANode.init` a boolean query function, named `isInitNode`, which returns `true` or `false` depending on whether a `Node` (n) is the initial node of the `ActivityGraph` to which it belongs.

Rule M_{10} maps `Edge` to an “edge” relation on `ANode`, named R_{edge}, that returns a set of `ANode` pairs (a_1, a_2), each of which qualifies to form the (source, target) pair of an `Edge`: a_2's `refCls` is one of the domain classes in a_1's `outClses`.

Finally, rules M_{11}–M_{14} map `AGraph` to `ActivityGraph`. Rules M_{11} and M_{12} map `ActivityGraph` to `AGraph` and `ActivityGraph.nodes` to `AGraph.nodes`, respectively. Rule M_{13} maps `ActivityGraph.edges`

Table 4: Rules for mapping CM $\rightarrow$ CM$_T$

M*Id*	**CM**	**CM$_T$**
M_1	`Decision`, `Join`, `ActName`, `State`	(same)
M_2	`ModuleAct`	`MAct`
M_3	`Node`	`ANode`.(excl.)`nodeType`, `outClses`, `init`
M_4	`DecisionNode`	`ANode`(`nodeType`=`Decision`)
M_5	`ForkNode`	`ANode`(`nodeType`=`Fork`)
M_6	`JoinNode`	`ANode`(`nodeType`=`Join`)
M_7	`MergeNode`	`ANode`(`nodeType`=`Merge`)
M_8	`outClses`: `Node` $\rightarrow \mathcal{P}$(`Class`) $(\forall n \in$ `Node`$.n.$`out` $\neq \emptyset)$. `outClses`$(n) = \{c \mid c = e.n_2.$`refCls`$,$ $e \in n.$`out`$\}$	`ANode.outClses`
M_9	`isInitNode`: `Node` $\rightarrow$ `Boolean` $(\forall n \in$ `Node`$)$. `isInitNode`$(n) = (n \in n.$`graph.n0`$)$	`ANode.init`
M_{10}	`Edge`	$R_{edge} \subseteq$ `ANode` $\times$ `ANode`, $R_{edge} = \{(a_1, a_2) \mid a_1, a_2 \in$ `ANode`, $a_1.$`outClses.length` > 0, $a_2.$`refCls` $\in a_1.$`outClses`$\}$
M_{11}	`ActivityGraph`	`AGraph`
M_{12}	`ActivityGraph.nodes`	`AGraph.nodes`
M_{13}	`ActivityGraph.edges`	`edges`: `AGraph` $\rightarrow$ `ANode` $\times$ `ANode` $(\forall g \in$ `AGraph`$)$. `edges`$(g) =$ $\{(a_1, a_2) \mid a_1, a_2 \in g.$`nodes`$, a_2.$`refCls` $\in a_1.$`outClses`$\}$
M_{14}	`ActivityGraph.n0`	`initNodes`: `AGraph` $\rightarrow \mathcal{P}$(`ANode`) $(\forall g \in$ `AGraph`$)$. `initNodes`$(g) =$ $\{a \mid a \in g.$`nodes`$, a.$`init` $=$ `true`$\}$

to a query function, named `edges`, over `AGraph`, which returns a set of node tuples (a_1, a_2) that form the edges. Rule M_{14} maps `ActivityGraph.n0` to another query function, named `initNodes` and is also over `AGraph`, which returns all the `ANode`s whose `init` = `true`.

From the language-engineering perspective, it is important to show that the mapping CM $\rightarrow$ CM$_T$ is bijective. Bijective mapping [20] means that CM$_T$ has the same information capacity as CM. Mathematically [21], this also means that the mapping has an inverse or 'backward' mapping. It is through this inverse mapping and an ASM that is derived from the CM$_T$ (discussed in the next section) that we can generate the activity graph of an AGC, written in the textual syntax of a host OOPL.

Theorem 1. *Mapping CM $\rightarrow$ CM$_T$ is bijective, i.e. for every CM's instance, there exists exactly one CM$_T$'s instance to which it is mapped.* □

Proof. Because each mapping rule in Table 4 is applied independently, if we can prove that each of them is bijective then the entire mapping CM $\rightarrow$ CM$_T$ is bijective. We provide below a brief proof of each mapping rule.

Rules M_1–M_2: These are trivially bijective, because each preserves the information capacity of the relevant CM's meta-concepts.

Rule M_3: This is bijective because it preserves the information capacity of `Node`. In particular, the additional OCL constraints that we introduced in `ANode` help preserve meanings of the data types of `Node.refCls` and `serviceCls`.

Rule M_4: This is bijective because of the following reasons. The node types captured in `NodeType` are unique. Thus, field `ANode.nodeType` helps group `ANode`s into different disjoint subsets. In particular, for every `DecisionNode` n, there exists exactly one `ANode`, whose `nodeType` = `Decision` and other fields are set to the corresponding fields of n.

Rules M_5–M_7: These are bijective by the arguments similar to the one presented above for M_4.

Rule M_8: This rule shows how a new field `ANode.outClses` is constructed from the function `outClses` over `Node`. To prove for this rule, we first assume there exists a derived field, `Node._outClses`, whose extent is defined by function `outClses`. Rule M_8 then becomes the mapping rule `Node._outClses` $\rightarrow$ `ANode.outClses`. This mapping rule is bijective by definition.

Rule M_9: This rule also involves constructing a new field (`ANode.init`), whose extent is defined by a function (`isInitNode`) over `Node`. The proof thus proceeds similarly to the proof presented above for rule M_8 (i.e. using a derived field).

Rule M_{10}: We prove that for every `Edge` $e = (n_1, n_2)$, there is (a) one and (b) only one pair of `ANodes` $(a_1, a_2) \in R_{edge}$, such that $n_1 \rightarrow a_1$ and $n_2 \rightarrow a_2$.

(a): First, we select a_1 from n_1 by applying a suitable combination of the mapping rules M_3–M_9. Now, by construction we have a_1.`outClses` = `outClses`(n_1). Because of `Edge` e we have: n_2.`refCls` $\in$ `outClses`(n_1). Thus, if we select a_2 from n_2 (by applying suitable mapping rules among M_3–M_9) then a_2.`refCls` = n_2.`refCls` $\in a_1$.`outClses`. In other words, $(a_1, a_2) \in R_{edge}$.

(b): The bijectiveness of the mapping rules M_3–M_9 help guarantee this, because there is exactly one a_1 (*resp.* a_2) s.t $n_1 \rightarrow a_1$ (*resp.* $n_2 \rightarrow a_2$).

Rule M_{11}: This is bijective by definition.

Rule M_{12}: This is bijective by definition.

Rule M_{13}: We prove this by assuming the existence of a derived field `AGraph._edges`, whose extent is defined by function `edges`. Then this mapping rule trivially becomes the bijective mapping between two fields: `ActivityGraph.edges` $\rightarrow$ `AGraph._edges`.

Rule M_{14}: Similarly, this is proved by first assuming the existence of a derived field `AGraph._initNodes`, whose extent is defined by function `initNodes`. Then, the mapping between two fields (`ActivityGraph.n0` $\rightarrow$ `AGraph._initNodes`) is a bijective mapping. □

5.3.3. The Annotation-Based ASM

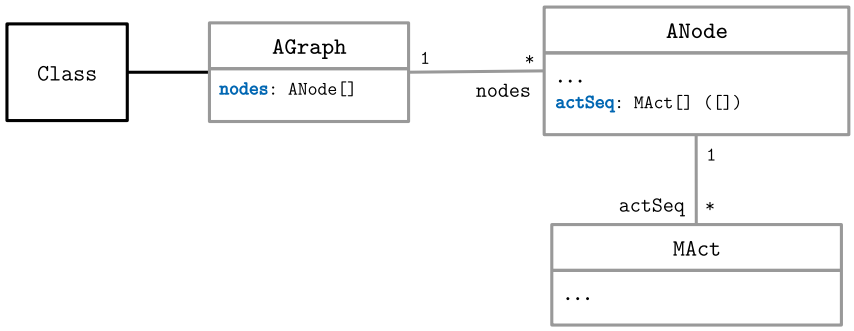

Figure 10: The annotation-based ASM.

Although CM_T is suitable for OOPL's representation, it is still not yet natively in that form. Our next step, therefore, is to transform it into an ASM that is "embedded" into OOPL. This ASM is constructed from the following 3 OOPL meta-concepts that were discussed in Section **??**: class, annotation and property.

Figure 10 shows the UML class diagram of ASM. In this, the three classes in CM_T are transformed into three annotations of the same name. The annotations are represented in the figure as 2-part grey-coloured boxes, the association lines as grey lines. Each domain field is transformed into an annotation property. The non-associative domain fields are transformed directly into properties and so, to ease reading, we use '...' to represent these properties. We only highlight in the figure two properties of the two associative fields `AGraph.nodes` and `ANode.actSeq`.

A key difference between ASM and CM_T is the attachment of `AGraph` to `Class`. This is represented in Figure 10 by a solid line connecting the two corresponding class boxes. An `AGraph` attachment defines an AGC because it describes the instantiation of an `AGraph` object together with the associated `ANodes` and `MAct` objects.

Adding the `AGraph` attachment to our definition of activity class (see Definition 1) helps form a bridge between AGL and the unified model. More specifically, in the overall context of our method, we call any

class that has an `AGraph` attachment an *activity class*. Further, to ease discussion we will use the term **configured unified model** to refer to a unified model whose the activity class is attached with an `AGraph`.

The following theorem ensures that AGL's ASM has the same information capacity as CM_T through the transformation.

Theorem 2. *The mapping $CM_T \rightarrow ASM$ is bijective.* □

Proof. The proof of Theorem 2 is trivial as it follows directly from the fact that the ASM does not contain any new features and from the following two properties about the mapping. First, each class in CM_T is bijectively mapped to one annotation in ASM. Second, each associative field in CM_T is bijectively mapped to an annotation property. □

5.3.4. Discussion

In the current syntax, the AGC is sensibly attached to the activity class, because this class serves as the pivot for the activity graph definition. An alternative annotation-based syntax would be to not define the `ANodes` as a property of `AGraph` (i.e. to remove property `AGraph.nodes`), but to distribute them such that they are attached to the domain classes that they reference (via the property `ANode.refCls`).

However, this syntax has several limitations. First, we need extra properties in order to keep track of which `ANodes` belong to which `AGraph`. For example, we need two new properties `AGraph.id` and `ANode.graph`, the values of which in the same `AGraph` are equal. Second, it is more difficult to read, understand, and validate the AGC. This is because the AGC is not in one place but is scattered around in different parts of the domain model. Third, we would unnecessarily complicate the component classes with `ANode` specifications, which in turn would hinder their use and understandability. These classes should only be concerned with the domain logics, not the mechanics of the activity graph that executes them.

5.4. Concrete Syntax Model (CSM)

Because ASM 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 [11] 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 exposition purpose in this paper, we will textually write an AGC using the structured note box notation of DCSL (explained in Section 2.2). The following example will help to illustrate.

Example: AGC and configured unified model

Figure 11 depicts the configured unified model of the enrolment management activity shown in Figure 8. As shown in Figure 11, the entire AGC is defined by an `AGraph` element, which is written within a note box attached to the activity class `EnrolmentMgmt` of the unified model. As can be seen from the figure, the `AGraph` element is configured with its property `nodes` being set to an array of four `ANodes`. These `ANodes` configure the four `Node` objects listed earlier in Table 3, and additionally for each of them the component class(es) that will become the referenced domain classes of the target nodes of the outgoing edges (if any). These component class(es) are specified by property `ANode.outClses`. For example, the first `ANode` configures the state of the node 1:`Node`. Property `outClses` of this `ANode` is set to the array [`DHelpOrSClass`], which states that 1:`Node` has an outgoing edge whose target node is the node whose *ref* domain class is `DHelpOrSClass`. According to Table 3 this is node 2:`Node`, and the outgoing edge is 1:`Edge`.

5.5. Semantics

Because CM, CM_T and the AGL's ASM have the same information capacity, we can discuss the AGL's semantics using any of these models. We choose CM because it has a clearer conceptual structure. Based on this structure (see Figure 7), we argue that the AGL's semantics is an extension of the core UML activity graph semantics to incorporate `ModuleAct` as a type of execution node. Indeed, Figure 7 shows that CM consists in `ModuleAct` (positioned at the top of the figure) and the UML activity graph, scoped by the

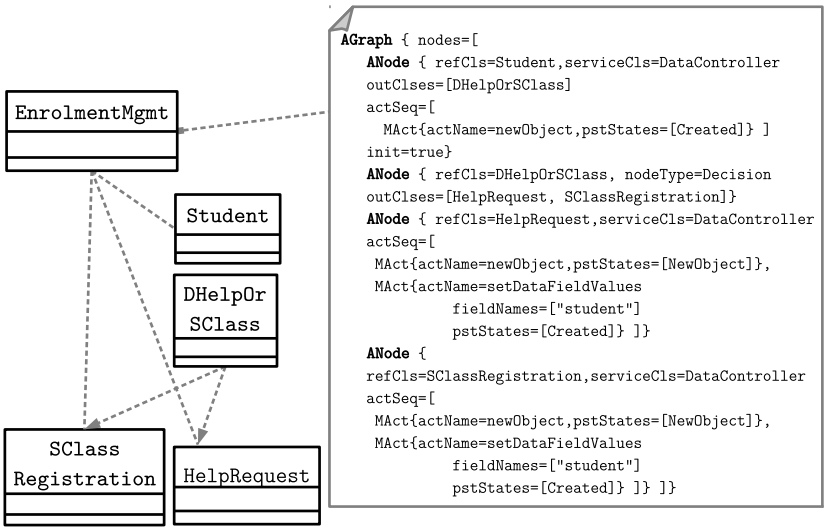

Figure 11: Configured unified model of the enrolment management activity: (LHS) the unified model, (RHS) the AGC written in the annotation-based concrete syntax.

inclusion, exclusion and restriction clauses in Section 5.1. The semantics of `ModuleAct` was discussed in Section 4, while the semantics of UML activity graph is defined informally in the UML specification [8] itself and formally in [22].

We conclude this section with an updated definition of the software generated in MOSA. This definition makes precise the general notion of module-based software that we introduced in Section 2.3 and takes into account the combination of unified model and 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 AGC describing the activity graph logic of an activity in the UML activity model 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 AGC attached to that class.* □

6. Tool Support

We implemented our method as a tool in a Java software framework that we reported in previous works [5, 7, 10]. The tool is available at `https://github.com/vnu-dse/jdomainapp-modules/tree/master/agcl`. It takes as input a configured unified model and semi-automatically generates as the output an interactive software prototype. This prototype is used by the development team to develop the domain model and, once this is completed, may also be reused to develop the production software.

Conceptually, the tool consists of three key components: model manager, view manager and object manager. First, **model manager** is responsible for registering the configured unified model and making it accessible to other components. Second, **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 of a set of object UIs (one for each module's view), and a desktop for organising these UIs. For example, Figure 12 shows the generated GUI for one variant of the CourseMan unified model. The GUI contains three object UIs for `ModuleEnrolmentMgmt`, `ModuleStudent`, and `ModuleEnrolment`. Later in the Section 7.1, we will demonstrate how the tool is used to generate several other variants of the CourseMan unified model.

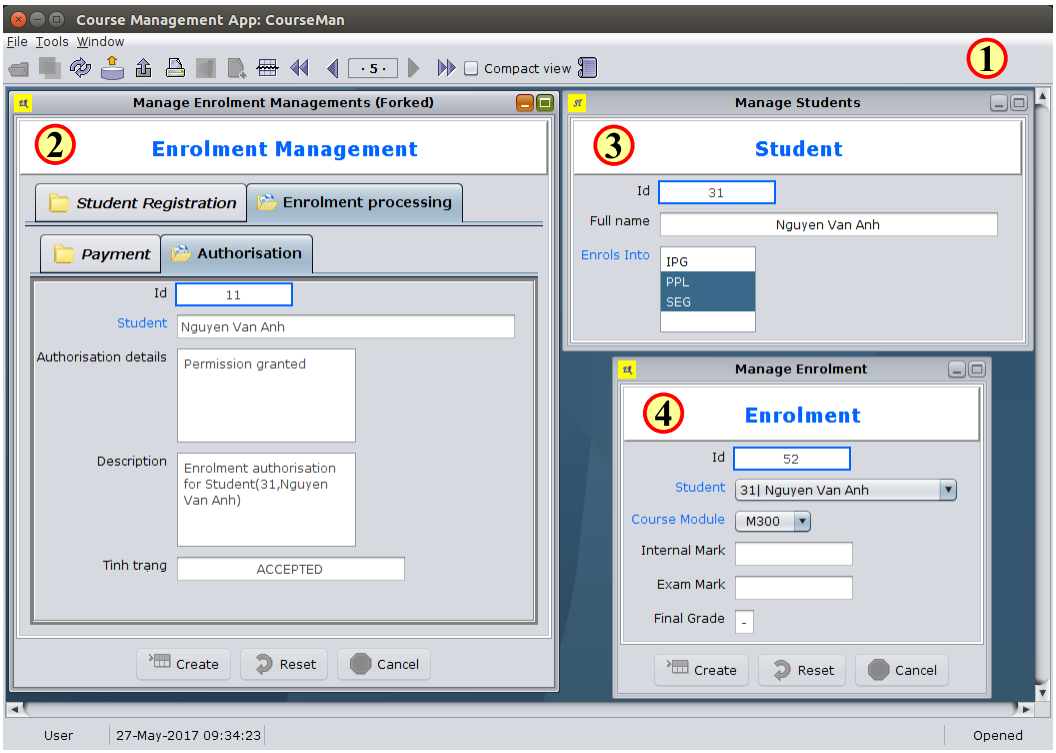

Figure 12: The GUI of CourseMan software generated by the tool: (1) desktop, (2-4) the object UIs of `EnrolmentMgmt`, `Student`, and `Enrolment`.

Third, **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 an 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.

7. Evaluation

In this section, we discuss an evaluation of AGL. Our aim is to show that AGL is both essentially expressive and practically usable. In this paper, we focus on evaluating AGL because it is a new language contribution of our method. We consider AGL as a type of specification language and adapt the DCSL evaluation approach that we applied in [7]. More specifically, we adapt from [23] the following three criteria for evaluating AGL: expressiveness, required coding level, and constructibility. We will present our evaluation of these criteria in Sections 7.1–7.3. We conclude in Section **??** with a number of discussion points concerning the evaluation.

7.1. Expressiveness

This is the extent to which a language is able to express the properties of interest of its domain [23]. For AGL, the domain properties are captured as meta-concepts and associations in the language's ASM. We measure the expressiveness of AGL by showing how it is able to express the five essential UML activity modeling patterns that we presented in [7]. We named the patterns after these five elementary activity flows: sequential, decisional, forked, joined and merged. Combinations of the patterns are used to build complete software. In this paper, we extend each pattern solution (presented in [7]) with an AGC in order to express the configured unified model.

We are particularly interested in the design of the *pattern form* [24, 25]. To keep the patterns generic, we present for each pattern form a UML activity model and a **template configured unified model** that realises it. The template model is a 'parameterised' configured unified model, in which elements of the nonannotation 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.

Similar to [7], we illustrate each pattern with a variant of the unified model for the enrolment management activity of CourseMan. A pattern example includes a configured unified model and one or more software GUIs. In this paper, we will focus on presenting the configured unified model and, in particular, its AGC. Please refer to [7, 26] for details about the unified model and the software GUI of each example. The CourseMan software of each example is automatically generated from the configured unified model, using the software tool described in Section 6.

7.1.1. Sequential Pattern Form

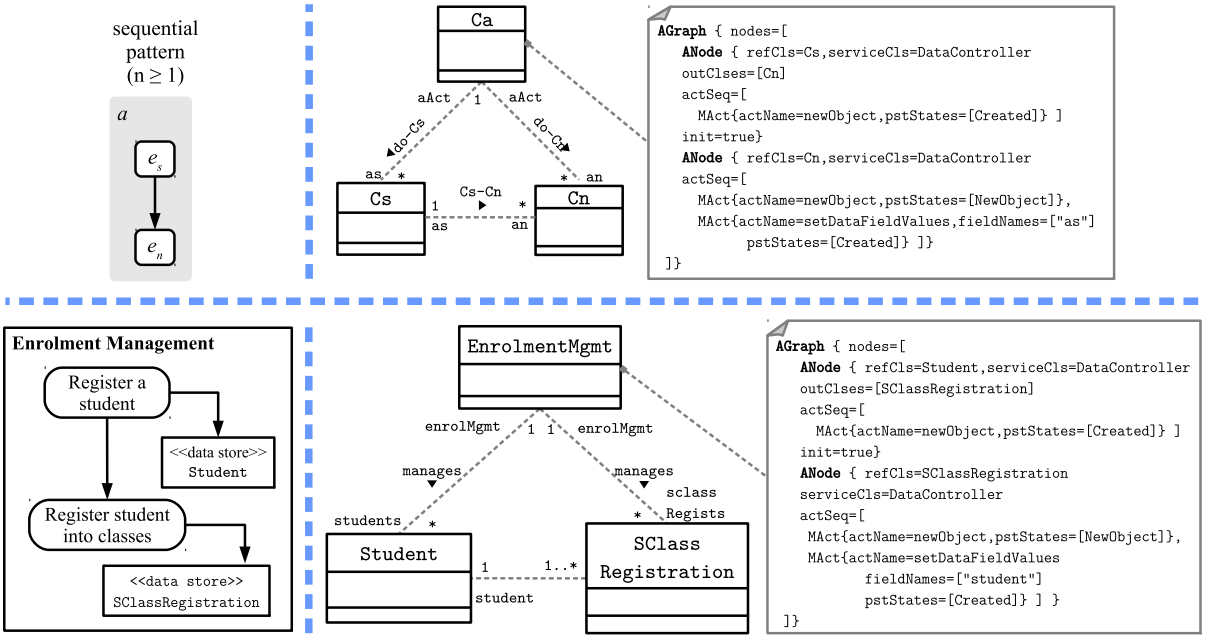

Figure 13: The sequential pattern form.

The top-left of Figure 13 shows the UML activity model, while the top-right shows the template configured unified model. This model consists of three classes `Ca`, `Cs`, and `Cn`. Class `Ca` is the activity class and has two associations with the two data classes `Cs` and `Cn`. These are the *ref* domain classes of the two action nodes e_s and e_n, *resp.*

The AGC is given in the `AGraph`'s note box in the top-right corner of the figure. It consists of two `ANodes`. The first `ANode` specifies node e_s, and the second specifies node e_n. The `ANodes` are quite self-explanatory except for the three `MActs`, which are worth some explanation. The first `MAct`'s configuration specifies the SAA (`newObject`, {`Created`})). This SAA is a subset of the one presented in Section 4.2.4. It involves performing (`newObject`, `NewObject`) and any combination of (`setDataFieldValues`, `Editing`) and (`createObject`, `Created`). Action `newObject` is to prepare `Cs`' view for user to enter input. Action `setDataFieldValues` is to set a view field's value from each user input (allowing user to re-enter if an error occurs). And action `createObject` is to create a new `Cs`' object from the input.

The second and third `MActs` together perform a similar logic over `Cn`, except for the need to break the `setDataFieldValues` operation into two steps: *(a)* set the `Cs` object created by the first `MAct` (and

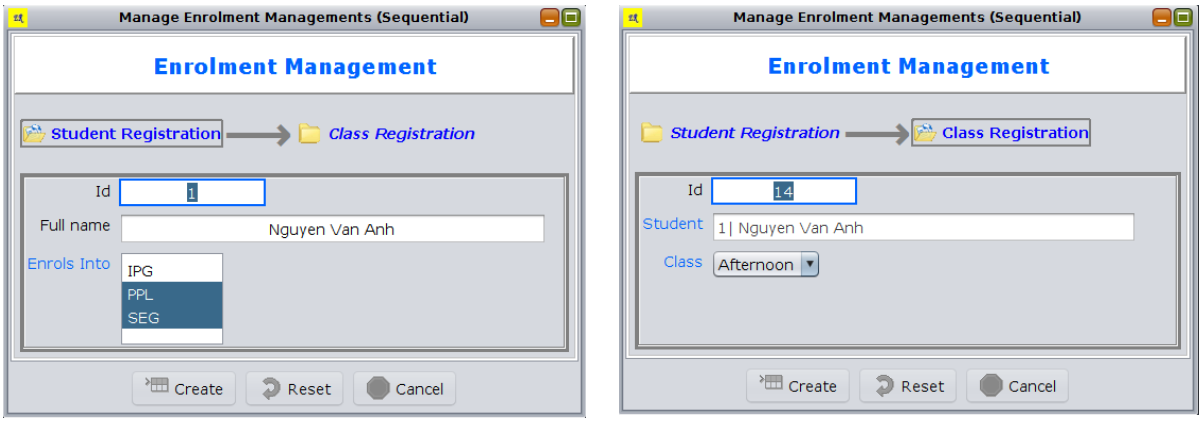

Figure 14: The sequential pattern form view of enrolment management activity.

offered by e_s to e_n via its output pin) into a suitable view field of `Cn`'s view and *(b)* set values of other view fields (allowing user to re-enter if an error occurs). To achieve this, the second `MAct` first specifies (`newObject`, {`NewObject`}). The third `MAct` then specifies the rest of the logic. Step *(a)* is performed by the operation `setDataFieldValues`, which uses the field named "`as`" to identify the view field of `Cn`'s view whose value needs to be set. Step *(b)* is performed by `setDataFieldValues` (as explained above) for other view fields.

Example

The bottom of Figure 13 shows how the pattern is applied to a simple variant of the CourseMan's enrolment management activity. The UML activity model involves performing two actions in sequence. The first action (`Student`) registers a student into course modules, while the second action (`SClassRegistration`) registers the student into a preferred class.

In this example: `Ca` = `EnrolmentMgmt`, `Cs` = `Student`, $n = 1$, `C1` = `SClassRegistration`.

The two GUI snapshots of the example are shown in Figure 14: one snapshot for the view of the one action. Each view is embedded in the `EnrolmentMgmt`'s view. The overall layout is a tab layout and the view of each associated module is contained in a tab of this layout. The left-hand-side figure shows the tab containing the `Student`'s view, while the right-hand-side one shows the tab containing the `SClassRegistration`'s view. Note, in particular, that the view field of the field `SClassRegistration.student` (field `Cn.as` in the template model) is automatically set to the `Student`(`name`="`Nguyen Van Anh`"), which is created on the `Student`'s view.

7.1.2. Decisional Pattern Form

The top-left of Figure 15 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.

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`.

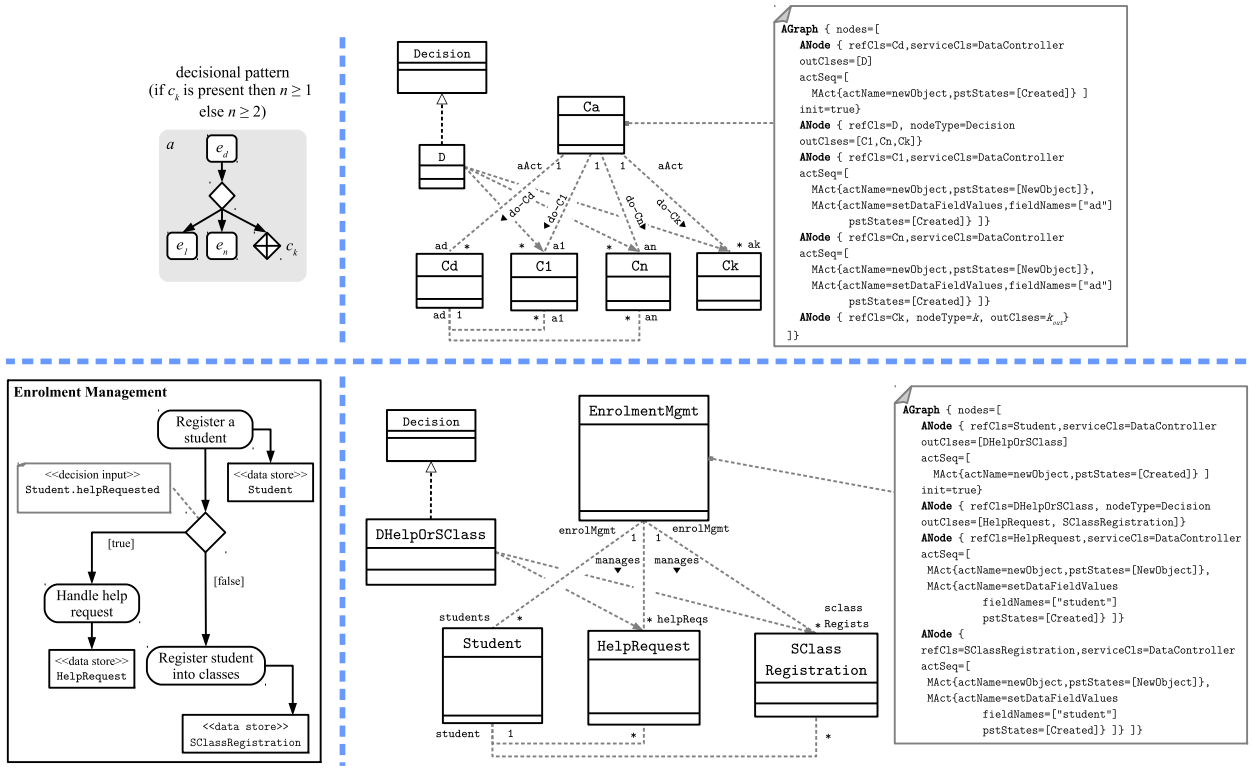

Figure 15: The decisional pattern form.

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 AGC consists of five `ANodes`. The first `ANode` is to create a new `Cd` object. The second `ANode` is to run the decision logic. The third and fourth `ANodes` represent the two decision cases: the first results in creating a new `C1` object for the specified `Cd` object, the second, 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 15 shows how the pattern is applied to the variant of CourseMan's enrolment management activity that we introduced in the example of Section **??**. The configured unified model, however, is a more detailed version of the one presented in Figures 5 and 11.

In this example: `Ca` = `EnrolmentMgmt`, `Cd` = `Student`, `D` = `DHelpOrSClass`, $n = 2$, `C1` = `HelpRequest`, `C2` = `SClassRegistration`. The control node c_k is not specified.

The three GUI snapshots of the example are shown in Figure 16. 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.*). Similar to the GUI of the sequential pattern, the activity's GUI contains the GUIs of the three actions in separate tabs. Under both cases of the decision, the `Student` object that is created in the first action (e.g. `Student`(`name`="Nguyen Van Anh")) is passed on to the next action. This object is then presented in the data field of the associative field `student` of the domain class referenced by this action.

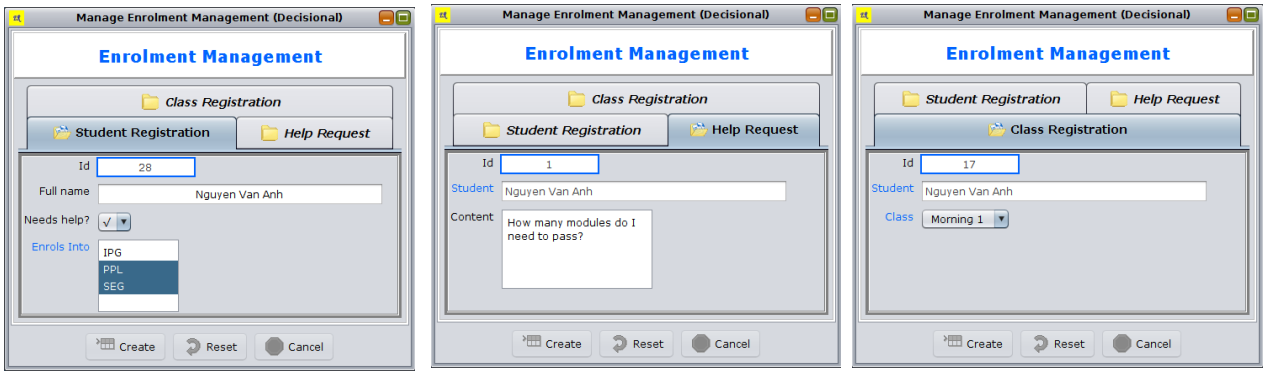

Figure 16: The decisional pattern form view of enrolment management.

7.1.3. Forked Pattern Form

The top-left of Figure 17 shows the UML activity model, while the top-right shows the template configured unified model. The activity class `Ca` has two associations to domain class `Cf` (referenced by node e_f) and control class `Co` (referenced by the forked node). Class `Co` in turn has associations to the other three domain classes, namely `C1`, `Cn`, and `Ck`. In addition to these, class `Cf` has two associations to `C1` and `Cn`, as these classes need to know `Cf` through object passing.

Similar to the decisional activity's template model, we consider the case that the class `Ck` is not a decision class. When this class is a decision class, we unfold it into the template model.

The AGC is very similar to that of the decisional activity's template model, except for in the second `ANode`: the reference class is `Co` (as opposed to `D`), the node type is `Fork` and property `serviceCls` is `DataController`. This property configuration makes explicit the fact that `Co`'s module service is available for use if needed.

Example

The bottom of Figure 17 shows how the pattern is applied to another variant of the CourseMan's enrolment management. The UML activity model of this variant involves performing student registration (e_f) and then two support actions concurrently: payment processing (e_1) and enrolment authorisation (e_2). These actions must be completed in order for any subsequent actions to proceed.

In this example: `Ca` = `EnrolmentMgmt`, `Cf` = `Student`, `Co` = `FEnrolmentProcessing`, n = 2, `C1` = `Payment`, `C2` = `Auhorisation`.

The three GUI snapshots of the example are shown in Figure 18: one snapshot for one action. The first snapshot is for the first action (student registration). The second and third snapshots are for the two concurrent actions (*resp.*): making payment and enrolment authorisation. A difference between this GUI and the GUIs of the previous two patterns is that it has a 2-level containment. This 2-level containment reflects the length-2 association chain from `EnrolmentMgmt` (the activity class) to `Payment` and `Authorisation`. The first-level containment is that between the activity's GUI and `Student`'s and enrolment processing's GUI. The second-level containment is that between enrolment processing's GUI and `Payment`'s and `Authorisation`'s GUI.

7.1.4. Joined Pattern Form

The top-left of Figure 19 shows the UML activity model, while the top-right shows the template configured unified model. The activity class `Ca` has associations to the domain classes `C1`, `Cn`, `Ck`, and `Cj`. These classes are referenced by the activity nodes e_1, e_n, c_k, and e_j (*resp.*). Class `Cj` has two associations to `C1` and `Cn`, as it knows these two classes through object passing. Class `J` implements the interface `Join` for the join logic.

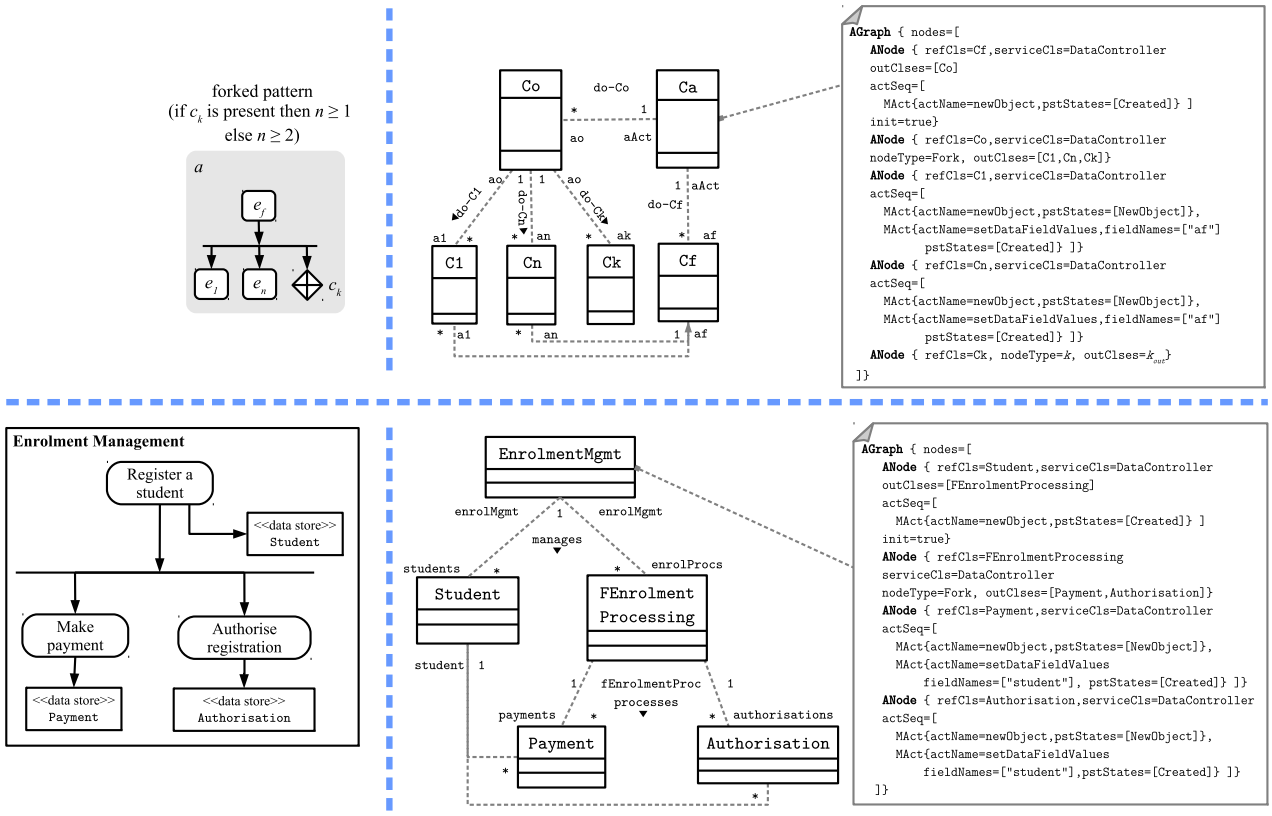

Figure 17: The forked pattern form.

Similar to the decisional activity's template model, we create three (optional) weak dependency associations between class `J` and three domain classes `C1`, `Cn`, and `Ck`. In addition, we consider the case that the class `Ck` is not a decision class. When this class is a decision class, we unfold it into the template model.

The AGC consists of five `ANodes`. The first three `ANodes` configure the three start nodes e_1, e_n, and c_k; and thus they all have `init=true` and `outClses` pointing to the domain class `J`. This class, which realises the interface `Join`, is referenced by the fourth `ANode`. This `ANode` configures the join node, and thus has `nodeType=Join`. The last `ANode` configures the last node e_j. It specifies two `MAct`, the second of which references the action `setDataFieldValues` which sets the view field values of the two fields "`a1`" and "`an`". The input for this operation is the output of the operation `Join.transf`.

Example

The bottom of Figure 19 shows how the pattern is applied to another variant of the CourseMan's enrolment management activity. The UML activity model of this variant involves joining two actions that concern the same `Student`, namely making payment (e_1) and registration authorisation (e_2), before concluding at the enrolment approval action. This last action decides, based on the results of the other two actions, whether or not to approve the student enrolment.

In this example: `Ca` = `EnrolmentMgmt`, $n = 2$, `C1` = `Payment`, `C2` = `Auhorisation`, `J` = `JPaymentAuthorise`, `Cj` = `EnrolmentApproval`. Note that in addition to the newly added associations in the example model, class `EnrolmentProcessing` is added with a domain field `student`, which is used to obtain input from the user for a `Student`. This `Student` object is needed to initialise the payment and authorisation processes.

The three GUI snapshots of the example are shown in Figure 20: the first snapshot is for making payment, the second is for registration authorisation, and the third is for enrolment approval.

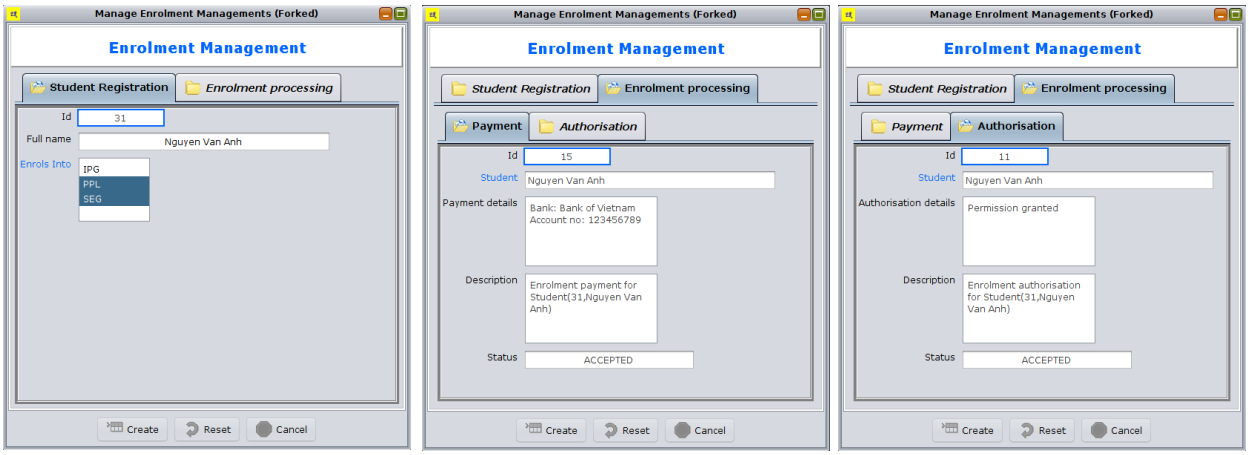

Figure 18: The forked pattern form view of enrolment management activity.

7.1.5. Merged Pattern Form

The top-left of Figure 21 shows the UML activity model, while the top-right shows the template configured unified model. The activity class `Ca` has associations to the domain classes `Cg` and `Cm`. These classes are referenced by the merge node and the activity node e_m (*resp.*). Class `Cg` has associations to the domain classes of the other nodes, namely `C1`, `Cn`, and `Ck`. Class `Cm` has associations to class `C1` and `Cn` as it knows these two classes through object passing.

Similar to the decisional activity's template model, we consider the case that class `Ck` is not a decision class. When this class is a decision class, we unfold it into the template model.

The AGC is very similar to that of the join pattern form, except for the configuration of the fourth `ANode`, which specifies that the node type be `Merge` rather than `Join`.

Example

The bottom of Figure 21 shows how the pattern is applied to another variant of the CourseMan's enrolment management activity. The UML activity model of this variant involves merging two actions, namely student class registration (e_1) and attending an orientation (e_2), to conclude at the action enrolment closure (e_m). The assumption here is that students can perform any combination of the two actions e_1, e_2. The completion of any one action will lead to enrolment closure. The action that has not yet been performed can be performed by the students at some later time.

Action e_2 requires adding a new domain class named `Orientation` to the CourseMan's domain model. This class is displayed at the bottom right of Figure 21. Thus, in this example: `Ca` = `EnrolmentMgmt`, $n = 2$, `C1` = `SClassRegistration`, `C2` = `Orientation`, `Cg` = `MgEnrolmentProcessing`, `Cm` = `EnrolmentClosure`.

The GUI snapshots of the example are shown in Figure 22. It shows a length-2 association chain from `EnrolmentMgmt` to `SClassRegistration` and `Orientation`. The first-level containment is that between the `EnrolmentMgmt`'s GUI and `MgEnrolmentProcessing`'s and `EnrolmentClosure`'s GUI. The second-level containment is that between `MgEnrolmentProcessing`'s GUI and `SClassRegistration`'s and `Orientation`'s GUI (the first and second snapshots). The `EnrolmentClosure`'s GUI is the third snapshot

7.2. Required Coding Level

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" [23]. Since AGL, to our knowledge, is the first aDSL of its type, we cannot compare AGL's RCL to other languages. Thus, we will measure the AGL's RCL using the "compactness" of the language's CSM (see Section 5.4). This is determined based on the reduction in the number of features in the *ASM* through

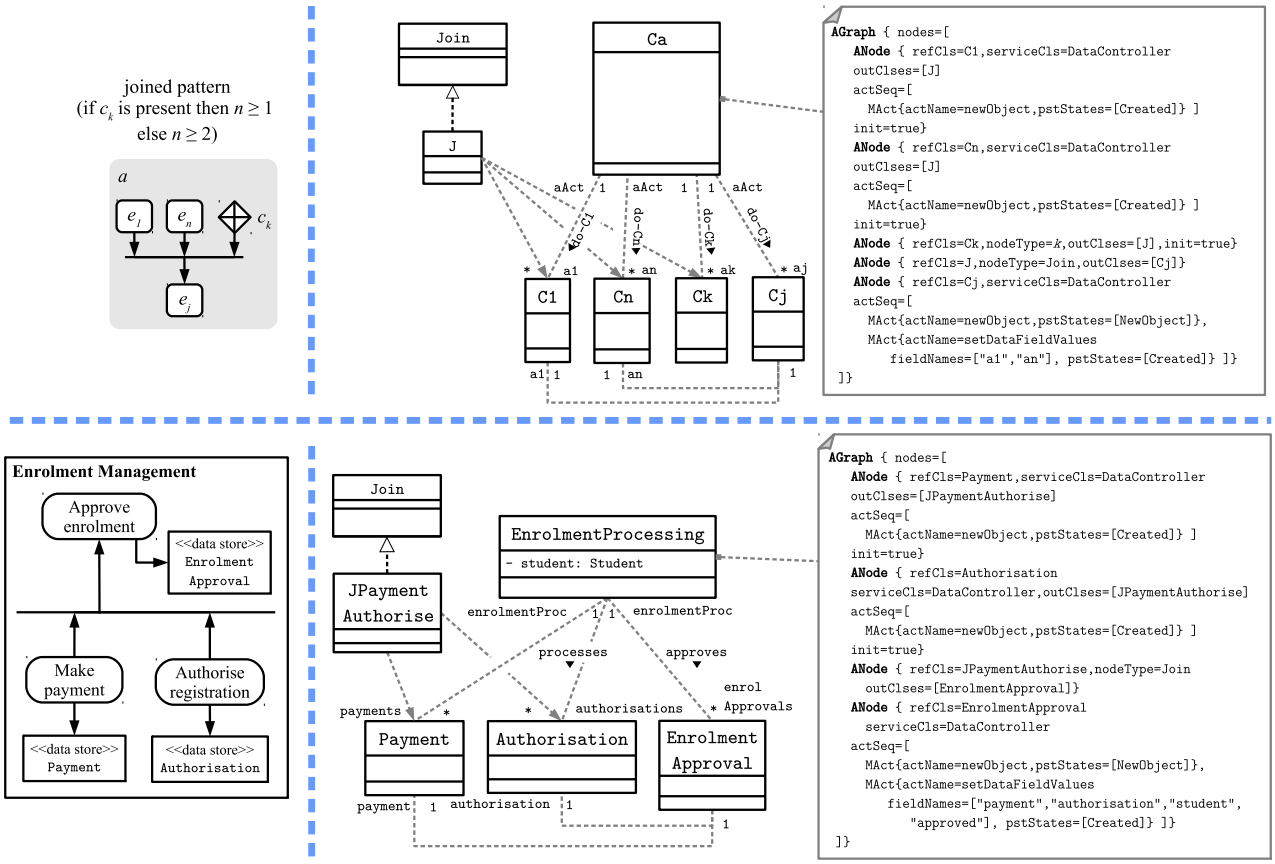

Figure 19: The joined pattern form.

the transformation CM $\rightarrow$ CM$_T$. More precisely, AGL's RCL is the percentage of the number of CM$_T$'s features over the number of CM'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 7 and 9 that AGL's RCL = $\frac{3}{9}$ or approximately 33%. Specifically, Figure 7 shows that the number of meta-concepts of the CM involved in the transformation is 9. These exclude the four meta-concepts (`ActName`, `State`, `Decision` and `Join`) that are transferred directly to CM$_T$. On the other hand, Figure 9 shows that 3 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.

7.3. Constructibility

This is the extent to which a language provides "...facilities for building complex specifications in a piecewise, incremental way"[23]. 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 5.2) that the nodes and edges of an activity graph are mapped to the domain classes and their associations.

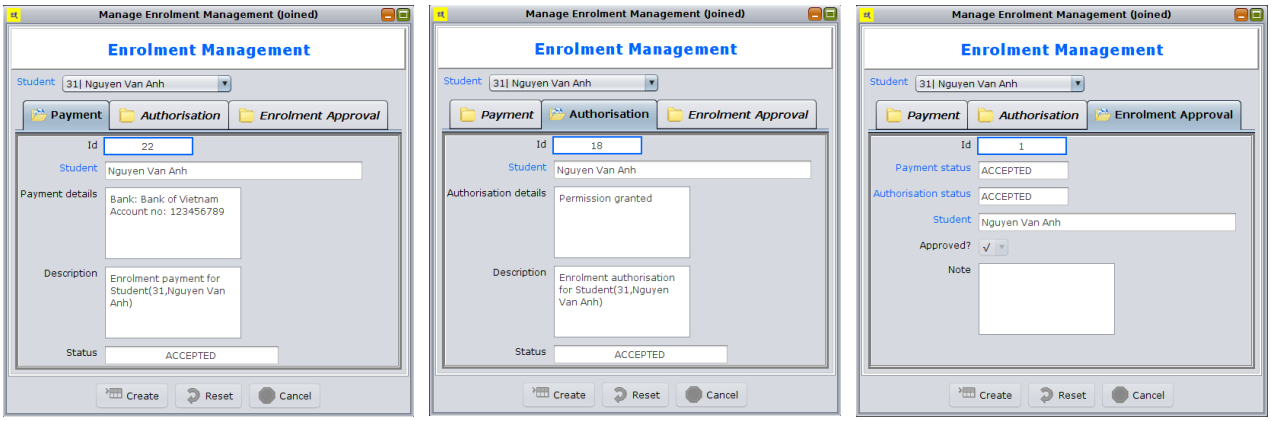

Figure 20: The joined pattern form view of enrolment management activity.

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 customise it. We plan to investigate techniques such as this as part of future work.

8. Threats to Validity

Discuss theats to validity of both our proposed method and the evaluation method.

8.1. The Proposed Method

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 [27] and agile [28] 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 feedbacks 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 [29, 30] to automatically generate our configured unified model from a high-level one that is constructed using a combination of UML class and activity diagrams.

Usability of the software GUI, from the domain expert's view-point, plays a role in the usability of our method. Although in this paper we did not discuss this issue, we would argue in favour of two aspects of the software GUI, namely simplicity and consistency, which contribute towards its learnability [31]. Our plan is to fully evaluate GUI usability in future work. First, the GUI design is simple because, as discussed in [7], 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.

8.2. Evaluation Method

The composition of the configured unified model in terms of the unified model and an activity graph model (see Section 5) follows a language composition approach described by Kleppe [11]. In this

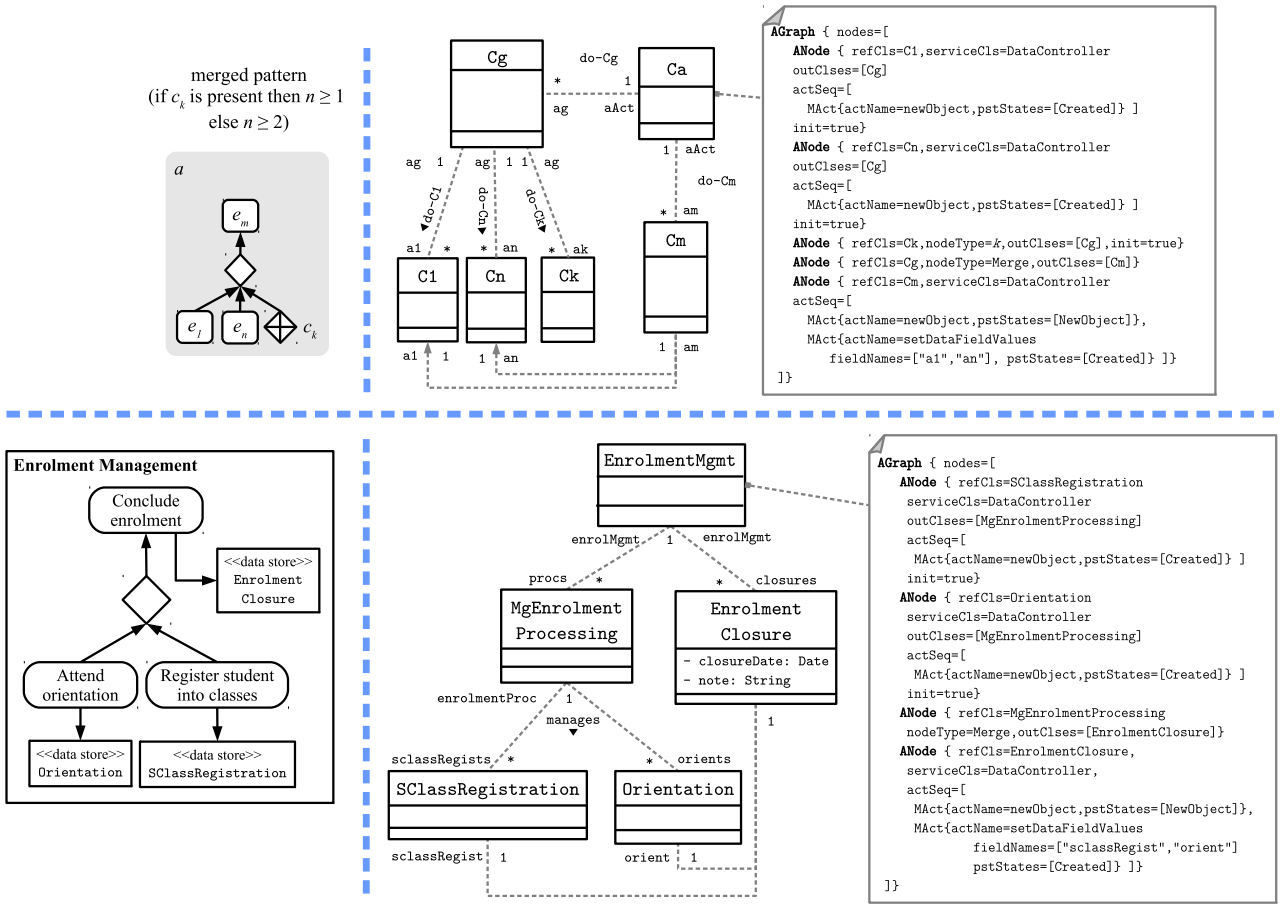

Figure 21: The merged pattern form.

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.

Evolution of languages (including both AGL and DCSL) is inevitable if we are to support more expressive domain modeling requirements. We discuss in [7] 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 (§15.6 [8]).

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. Related Work

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

DSL Engineering. DSLs [32, 33] can be classified based on domain [11], as vertical or horizontal, or based on the relationship with a host language [14, 32, 33], as internal or external. Our proposed AGL is a

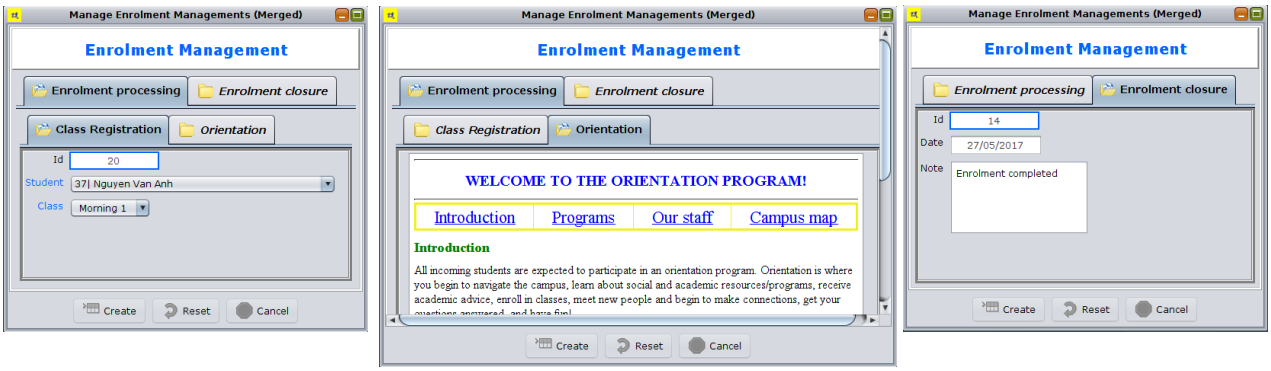

Figure 22: The merged pattern form view of enrolment management activity.

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 [14] by both the DDD's author and others. However, the work in [14] 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.*).

Behavioural modeling with UML activity diagram. Although in his book [1] Evans does not explicitly mention behavioural 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. For example, in Chapter 2 of the book, when discussing the use of documents and diagrams (i.e. models) in the ubiquitous language, Evans states the followings:

- "the attributes and relationships are only half the story of an object model. But the behavior of those objects and the constraints on them are not so easily illustrated. Object interaction diagrams can illustrate some tricky hotspots in the design..."
- "Behavioral responsibilities of an object can be hinted at through operations names, and implicitly demonstrated with object interaction (or sequence) diagrams."

In UML [8] (§13.2.1), interaction diagrams (such as sequence diagram) are only one of three main diagram types that are used to model the system behavior. The other two types are state machine (§14) and activity diagram (§15, 16). Although in the book, Evans only uses sequence diagram 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 (§16) 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 behavioural modeling method. Our combination of two aDSLs in this paper helps fills this gap.

Our definition of module action in this paper incorporates the notion of state, which is more formally modelled in another UML behavioural modeling language called Behavior State Machines (BSM) (§14.2 [8]). The reason that this is possible is because Activity diagram and BSM are tightly linked insofar as behavior is tightly linked to state and state transition. Indeed, these languages represent two sides of the same coin: the former emphasises the actual behavior, while the latter focuses on the behavior's effects (states and state transitions). More specifically, a close inspection of the BSM's abstract syntax (§14.2.2) reveals that both `State` and `Transition` have associated `Behavior`(s) that describe what actually takes place when a

particular state is reached or during a transition between some two states. Our notion of module action's pre- and post-states simply looks at the same view but from the behavior's perspective.

Unified modeling with UML diagrams. There have been works attempting to combine UML structural and behavioural 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 [34, 35] discuss combining UML class and state machine diagrams to model the system. Another work [36] 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.

MVC architecture. In practical software development, the MVC (or other equivalent) 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 [37], due primarily to the human factors that are involved in the development process. MVC is considered in [38] to be one of several so-called agent-based design architectures, which help make software developed in them inherently modular and thus easier to maintain. Software that are 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 [39]), thereby creating a hierarchical design architecture in which a software is composed from a hierarchy of software modules. A software *module* (called PAC object in [39] and, more generally, agent in [38]) is a realisation of a coherent subset of the software's functions in terms of the architectural components.

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 [11, 30]. 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.

AtOP. Our idea of using annotation to represent modeling rules and constraints is inspired by AtOP [42–45]. In principle, AtOP extends a conventional program with a set of attributes, which capture application- or domain-specific semantics [43]. These attributes are represented in contemporary OOPLs as annotations.

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 [42]. More recently, the work in [45] 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 [42, 45] 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.

10. 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 UML activity diagram. It is expressed in DCSL, which is an aDSL that we developed in a previous work. To use the unified model at the core layer of MOSA, we developed another aDSL named AGL to construct for the model an UML activity graph. 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 in two important fronts: bridging the gaps between model and code and constructing a unified domain model. Our proposed aDSLs are horizontal DSLs which 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.

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, 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.

[5] D. M. Le, D.-H. Dang, V.-H. Nguyen, Generative Software Module Development: A Domain-Driven Design Perspective, in: Proc. 9th Int. Conf. on Knowledge and Systems Engineering (KSE), 2017, pp. 77–82.

[6] D. M. Le, D.-H. Dang, V.-H. Nguyen, Generative Software Module Development for Domain-Driven Design with Annotation-Based Domain
(Submitted to) Journal of Information and Software Technology.
URL https://drive.google.com/open?id=1v-A-v92Uo2hMPNtgY4seeYoijAf0wI9y

[7] D. M. Le, D.-H. Dang, V.-H. Nguyen, On Domain Driven Design Using Annotation-Based Domain Specific Language, Computer Languages, Systems & Structuresdoi:10.1016/j.cl.2018.05.001.
URL https://www.sciencedirect.com/science/article/pii/S147784241730204X

[8] OMG, Unified Modeling Language version 2.5 (2015).

[9] M. Dumas, A. H. M. t. Hofstede, UML Activity Diagrams as a Workflow Specification Language, in: M. Gogolla, C. Kobryn (Eds.), UML 2001, LNCS, Springer, 2001, pp. 76–90.

[10] D. M. Le, jDomainApp version 5.0: A Java Domain-Driven Software Development Framework, Tech. rep., Hanoi University (2017).

[11] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2008.

[12] OMG, Object Constraint Language Version 2.4 (2014).

[13] M. Nosál', M. Sulír, J. Juhár, Language Composition Using Source Code Annotations, Computer Science and Information Systems 13 (3) (2016) 707–729.

[14] M. Fowler, T. White, Domain-Specific Languages, Addison-Wesley Professional, 2010.

[15] 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.

[16] A. Hejlsberg, M. Torgersen, S. Wiltamuth, P. Golde, The C# Programming Language, 4th Edition, Addison Wesley, Upper Saddle River, NJ, 2010.

[17] V. Vernon, Implementing Domain-Driven Design, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2013.

[18] 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.

[19] G. Booch, Object-Oriented Development, IEEE Transactions on Software Engineering SE-12 (2) (1986) 211–221. doi:10.1109/TSE.1986.6312937.

[20] P. Stevens, A Landscape of Bidirectional Model Transformations, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 408–424. doi:10.1007/978-3-540-88643-3_10.
URL https://doi.org/10.1007/978-3-540-88643-3_10

[21] E. W. Weisstein, Bijective (2018).
URL http://mathworld.wolfram.com/Bijective.html

[22] Z. Daw, R. Cleaveland, An Extensible Operational Semantics for UML Activity Diagrams, in: R. Calinescu, B. Rumpe (Eds.), Software Engineering and Formal Methods, no. 9276 in Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 360–368. doi:10.1007/978-3-319-22969-0_25.
URL http://link.springer.com/chapter/10.1007/978-3-319-22969-0_25

[23] 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.

[24] D. Riehle, H. Züllighoven, Understanding and Using Patterns in Software Development, Theory Pract. Obj. Syst. 2 (1) (1996) 3–13.

[25] 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.

[26] D. M. Le, D.-H. Dang, V.-H. Nguyen, On Domain Driven Design Using Annotation-Based Domain Specific Language [Extended], Tech. rep., VNU University of Engineering and Technology (2017).
URL https://drive.google.com/open?id=1536zs321O66sR2azLgTU_pifjJY7sK-2

[27] 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.

[28] 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.
URL http://agilemanifesto.org/

[29] 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.
URL http://link.springer.com/chapter/10.1007/3-540-47884-1_16

[30] M. Brambilla, J. Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice, 1st Edition, Morgan & Claypool Publishers, 2012.
[31] 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.
URL http://www.sciencedirect.com/science/article/pii/S0164121202001590
[32] A. van Deursen, P. Klint, J. Visser, Domain-specific Languages: An Annotated Bibliography, SIGPLAN Not. 35 (6) (2000) 26–36.
[33] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[34] 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.
URL http://doi.acm.org/10.1145/337180.337207
[35] 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.
[36] P. Selonen, K. Koskimies, M. Sakkinen, Transformations Between UML Diagrams, JDM 14 (3) (2003) 37–55. doi:10.4018/jdm.2003070103.
URL https://www.igi-global.com/article/transformations-between-uml-diagrams/3298
[37] 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.
URL http://doi.acm.org/10.1145/2593882.2593883
[38] 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.
[39] J. Coutaz, PAC: An Object Oriented Model for Dialog Design, in: Interact'87, Vol. 87, Elsevier, 1987, pp. 431–436.
[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).
[42] 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.
[43] V. Cepa, S. Kloppenburg, Representing Explicit Attributes in UML, in: 7th Int. Workshop on AOM, 2005.
[44] M. Sulír, M. Nosáľ, J. Porubän, Recording Concerns in Source Code Using Annotations, Computer Languages, Systems & Structures 46 (2016) 44–65.
[45] 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).

Appendix A. Well-formedness Rules of AGL

Appendix A.1. `ActivityGraph`

```
1 -- nodes must contain at least two Nodes
2 context ActivityGraph inv:
3   not(nodes.oclIsUndefined()) and
4   nodes->size()>=2
```

```
1 -- edges must contain at least one Edge
2 context ActivityGraph inv:
3   not(edges.oclIsUndefined()) and
4   edges->size()>=1
```

```
1 -- n0 must contain at least one Node and
2 -- n0 is a subset of nodes
3 context ActivityGraph inv:
4   not(n0.oclIsUndefined()) and
5   nodes->size()>=1 and nodes->includesAll(n0)
```

```
1 -- (Connectedness) graph is connected (i.e.
2 -- at least one path connects any two nodes)
3 context ActivityGraph inv:
4   nodes->forAll(n, n' | hasPath(n, n'))
```

```
context ActivityGraph
-- determines if there is a path
-- (i.e. a sequence of edges) that
-- connect n to n' (in either direction)
def : hasPath(n : Node, n' : Node) : Boolean =
  edges->exists((n1 = n and n2 = n') or
                  (n1 = n' and n2 = n))
  or
  nodes->exists(n'' | hasPath(n,n'') and hasPath(n'',n'))
```

Appendix A.2. Node

```
-- refCls is a domain class
context Node inv:
  not(refCls.oclIsUndefined()) implies refCls.isDomainClass()
```

```
-- JoinNode.refCls must be a sub-type of Join
context JoinNode inv:
  refCls.oclIsTypeOf(Join)

-- DecisionNode.refCls must be a sub-type of Decision
context DecisionNode inv:
  refCls.oclIsTypeOf(Decision)

-- ForkNode.refCls must not be specified
context ForkNode inv:
  refCls.oclIsUndefined()

-- MergeNode.refCls must not be specified
context MergeNode inv:
  refCls.oclIsUndefined()
```

```
-- serviceCls (if specified) must contains a corresponding method
-- for every ModuleAct in actSeq
context Node inv:
  not(actSeq->isEmpty()) implies not(serviceCls.oclIsUndefined()) and
    actSeq->forAll(m | serviceCls.methods->exists(t | t.name = m.actName.value))
```

```
-- actSeq of ControlNode is undefined, while actSeq of non-ControlNode must not
    be empty
context Node inv:
  if oclIsTypeOf(ControlNode)
  then
    actSeq.oclIsUndefined()
  else
    not(actSeq.oclIsUndefined()) and not(actSeq->isEmpty())
  endif
```

Appendix A.3. Edge

```
-- n1 and n2 must be defined and
-- belong to graph.nodes
context Edge inv:
  not(n1.oclIsUndefined()) and
  not(n2.oclIsUndefined()) and
  graph.nodes->includes(n1) and
  graph.nodes->includes(n2)
```

Appendix A.4. ModuleAct

```
-- actName, postStates must be defined
context ModuleAct inv:
  not(actName.oclIsUndefined()) and
  not(postStates.oclIsUndefined()) and not(postStates->isEmpty())
```

```
-- fieldNames must contain unique elements, be length-equal to fieldVals and
-- contain names of domain fields of node.refCls
context ModuleAct inv:
  (not(fieldVals.oclIsUndefined() and fieldVals->isEmpty()) implies
    not(fieldNames.oclIsUndefined()) and fieldNames->size() = fieldVals->size())
  and
  (not(fieldNames.oclIsUndefined()) implies
    fieldNames->asSet() = fieldNames  -- isDistinct(fieldNames)
    and
    fieldNames->forAll(a | node.refCls.domFields()->exists(f | f.name = a)))
```

Appendix A.5. `JoinNode`

```
-- if pre is defined then pre contains only
-- the Nodes that are the source (n) of
-- some Edges (n,self)
context JoinNode inv:
  not(pre.oclIsUndefined()) implies
  pre->forAll(n | graph.edges->exists(e |
    e.n1 = n and e.n2 = self))
```

Appendix A.6. `Class`

```
context Class
  -- whether or not this is a domain class
  -- (i.e. is defined with DClass)
  def: isDomainClass() : Boolean =
    not(dcl.oclIsUndefined())

  -- if this contains domain fields then
  -- return them as Set else return {}
  def: domFields() : Set(Field) =
  fields->select(f |
    not(f.atr.oclIsUndefined()))
```