Information and Software Technology 120 (2020) 106239

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Generative software module development for domain-driven design with annotation-based domain specific language

Duc Minh Le[a,b,1], Duc-Hanh Dang[a,1], Viet-Ha Nguyen[a,*]

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

ARTICLE INFO

Keywords:
Domain-driven design (DDD)
Module-based architecture
UML-based domain modelling
Domain-specific language (DSL)
Object-oriented programming language (OOPL)
Attribute-oriented programming (AtOP)

ABSTRACT

Context: Object-oriented domain-driven design (DDD) aims to iteratively develop software around a realistic model of the application domain, which both thoroughly captures the domain requirements and is technically feasible for implementation. The main focus of recent work in DDD has been on using a form of annotation-based domain specific language (aDSL), internal to an object-oriented programming language, to build the domain model. However, these work do not consider software modules as first-class objects and thus lack a method for their development.

Objective: In this paper, we tackle software module development with the DDD method by adopting a generative approach that uses aDSL. To achieve this, we first extend a previous work on module-based software architecture with three enhancements that make it amenable to generative development. We then treat module configurations as first-class objects and define an aDSL, named MCCL, to express module configuration classes. To improve productivity, we define function MCCGen to automatically generate each configuration class from the module's domain class.

Method: We define our method as a refinement of an aDSL-based software development method from a previous work. We apply meta-modelling with UML/OCL to define MCCL and implement MCCL in a Java software framework. We evaluate the applicability of our method using a case study and formally define an evaluation framework for module generativity. We also analyse the correctness and performance of function MCCGen.

Results: MCCL is an aDSL for module configurations. Our evaluation shows MCCL is applicable to complex problem domains. Further, the MCCs and software modules can be generated with a high and quantifiable degree of automation.

Conclusion: Our method bridges an important gap in DDD with a software module development method that uses a novel aDSL with a module-based software architecture and a generative technique for module configuration.

1. Introduction

Object-oriented domain-driven design (DDD) [1,2] aims to develop complex software (iteratively) around a realistic domain model, which both thoroughly captures the domain requirements and is technically feasible for implementation. This requires close collaboration among all the stakeholders (including the domain experts and the technical team members), using a ubiquitous language [1] to construct the domain model (that satisfies the domain requirements) and resulting a object oriented implementation of this model. To achieve this, the DDD method generally suggests to use a conceptual layered software architecture, which includes the domain model at the core layer and other architectural concerns (e.g. user interface (UI), data storage *etc.*) being realised in other layers surrounding this core. In practice, complex software requires a scalable architecture and a highly productive software development method. The challenge for DDD, therefore, is how to construct the software using a scalable layered software architecture and with high productivity.

The state of the art solutions have only partially addressed this challenge. First, object oriented software design methods [3–5] generally suggest that scalable software also require a modular structure. According to Meyer [3], *software module* is a self-contained decomposition unit of a software. However, these methods are generic and thus they do not address the specific features of DDD mentioned above. Second, in his

* Corresponding author.
E-mail addresses: duclm@hanu.edu.vn (D.M. Le), hanhdd@vnu.edu.vn (D.-H. Dang), hanv@vnu.edu.vn (V.-H. Nguyen).
[1] Authors Le and Dang are co-first authors and contributed equally to leading this paper.

https://doi.org/10.1016/j.infsof.2019.106239
Received 7 August 2018; Received in revised form 20 September 2019; Accepted 26 November 2019
Available online 30 November 2019
0950-5849/

book [1], Evans only concentrates on scaling the domain model with the notion of *domain module*[2] A domain module is a logical grouping of a set of functionally-related domain classes. In object oriented design terminology [6], a domain module is a package. Current DDD software frameworks (ApacheIsIs [7] and OpenXava [8]) have taken this a little further by using a simple form of internal DSL to build the domain model and using this model as input to generate a software prototype. The internal DSL that they use is more formally known as *annotation-based domain specific language* (aDSL) [9,10], which is internal to an object-oriented programming language (OOPL). The generated software prototype has a UI for each domain class. However, these DDD works have not considered software modules in their design methods and, as such, they have not addressed how to construct software modules from the domain model.

In this paper, we propose to tackle the stated DDD challenge with a method to construct software modules directly from the domain model and to automatically generate these modules. We define our method as a refinement of an aDSL-based software development method for DDD that we proposed in a recent work [11]. We base our method on a module-based software architecture (MOSA), which we developed in other previous works [11–13]. In MOSA, we consider software module as a 'micro-software' consisting in a domain class (the core) and two other components, a view and a controller [14], that provide a UI for presenting the domain objects. Regarding to module generation, we treat it as a generative software development problem [15] – a module is generated from a set of reusable components using a *configuration*. In our method, the reusable components include a domain class, a view and a controller; the configuration is named *module configuration*. To construct this configuration, we consider it as object of *module configuration class* (MCC). We express the MCCs by an aDSL, named *Module Configuration Class Language* (MCCL). The benefit of using aDSL is that it allows both the software modules and the domain model to be expressed in the same host OOPL, which in turn significantly eases software construction from them. Further, to help improve development productivity in MCC construction, we define a generator function, named MCCGen, that automatically generates an MCC from a domain class.

In brief, in this paper we make the following contributions:

1. a **generative software module development method** for DDD, with MOSA and using aDSL. Our method consists in the components described below.
2. three **enhancements to module configuration** in MOSA that make it amenable to generative development.
3. an aDSL, named **MCCL**, for configuring software modules directly in a host OOPL.
4. function **MCCGen** that automatically generates MCC from a domain class.
5. a **software tool** to support the method that is implemented as part of a Java software framework.
6. an **evaluation** of the method through a case study, a formal evaluation framework for module generativity and a formal analysis of MCCGen. The evaluation shows that our method is applicable to developing complex software and that this can be performed with very high productivity.

Compared to our previous conference paper [13], this work adds three significant new contents (1,2,6) and includes in (3) a complete specification of the MCCL syntax using UML and OCL. The rest of the paper is structured as follow. Section 2 presents the background of our work. Section 3 gives an overview of our method. Section 4 discusses the MOSA enhancements. Section 5 specifies MCCL as a language. Section 6 presents the MCC generation technique. Section 7 discusses tool support for our method. Section 8 presents the evaluation of our method. Section 9 reviews the related work and Section 10 concludes the paper.

2. Background

In this section, we review a number of background concepts that will be used in the remainder of the paper. We will illustrate the concepts using a course management (CourseMan) software example. We adapted this example from Le [12] and extended it (in light of [16]) to allow many-many associations between domain classes.

2.1. A module-based software architecture for DDD

In this paper, we adopted a module-based software architecture (abbreviated in this paper as **MOSA**), which we developed in a previous work [12]. We chose MOSA because (*i*) it was developed based on the MVC architecture [14], (*ii*) it realises the DDD's layered architecture [1] and (*iii*) it is inherently module-based. MOSA positions the domain model at the core layer and (currently) incorporates the UI concern into a layer around this core. A software module in MOSA forms a micro, functional software. The modules can be combined to form a complete software. MOSA is built upon two key concepts: MVC-based module class and module containment tree.

Module class is a structured class [6] that describes the structure of modules in terms of three MVC components: model (a domain class), a view class and a controller class. More formally, given a domain class C, the view and controller classes are the parameterised classes `View`$\langle T \rightarrow C \rangle$ and `Controller`$\langle T \rightarrow C \rangle$ (*resp.*); where `View`$\langle T \rangle$ and `Controller`$\langle T \rangle$ are two library template classes, T is the template parameter. To ease discussion, we name a module class after its domain class, using the prefix

}}`Module`″; e.g. `ModuleStudent` is a module class created from `Student`. Further, we use the notation `View`$\langle C \rangle$ to denote the view of a `Module`C.

Every module class has a **state scope**, which is a sub-set of the attributes of the domain class that are relevant to the module. Relevancy is determined based on the module's specification. Module classes in a MOSA model are associated by virtue of the fact that their domain classes are associated in the domain model. These associations form the basis for module containment and containment tree. *Module containment* is a relation on the set of modules, while containment tree results from the structure of the module containments of a composite module. *Containment tree* is the basis for combining modules in MOSA. Given two domain classes C and D, a module $M =$ `Module`C (called **composite module**) contains another module $N =$ `Module`D (called **child module**), if (*i*) C participates in a binary association (denoted by A) with D, (*ii*) C is at the one end (if A is one-many), at the mandatory one end (if A is one-one) or at either end (if otherwise), and (*iii*) exists a subset of N's state scope, called **containment scope**, that satisfies M's specification.

Containment tree is a rooted tree whose root node is a composite module (called the **root module**). Other nodes are modules that are contained directly or indirectly by the root module. We call these the **descendant modules**. A tree edge from a parent node to a child node represents a module containment.

Example 1. The bottom of Fig. 1 is the **domain model** of CourseMan. The domain classes are created in a package named `courseman.model`. For brevity, we exclude attributes and operations from the class boxes. Class `Student` represents the domain concept Student[3] Class `CourseModule` represents the Course Modules[4] that are offered to students. We specifically label the `Student`'s end of the many-many association between this class and `CourseModule` with the role "modules". Class `SClass` represents types of classes (e.g. morning, afternoon and

[2] we use this term instead of Evans's "module" to make a distinction with *software module*.

[3] we use `fixed font` for model elements and normal font for concepts.

[4] we use class/concept name as countable noun to identify instances.

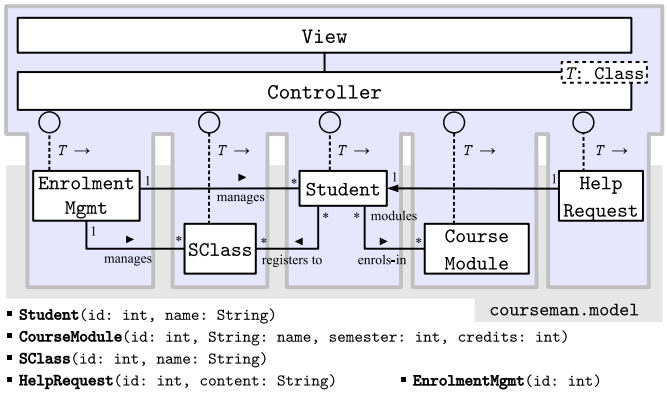

Fig. 1. The MOSA model of CourseMan.

evening) that Students can choose. Class EnrolmentMgmt represents the enrolment management activity. This involves registering Students, enrolling them into CourseModules and registering them into SClasses. In addition, it allows each Student to raise a HelpRequest.

The entire Fig. 1 is a MOSA model of CourseMan, which consists of four module classes. The model is drawn using the graphical notation described in [12]. A binding is depicted by a dashed line, whose Controller's end is drawn with the symbol '◯'. Among the module classes shown in the figure, ModuleEnrolmentMgmt is a composite module, whose containment tree first contains a child module ModuleStudent (because of the one-many association from EnrolmentMgmt to Student). This child module would be the parent of a child module typed ModuleCourseModule, if Student is preferred to CourseModule over the many-many association between them.

The state scope of ModuleStudent includes two attributes Student.id, name and another attribute, named modules, that realises the Student's role in the many-many association to CourseModule. The containment scope of ModuleStudent in the containment tree of ModuleEnrolmentMgmt is the same as its state scope. In the containment trees of other modules (e.g. ModuleClass's), however, ModuleStudent's state scope may contain only a proper subset of the state scope. □

Module configuration method

In [12], we view module configuration at two levels. At the *module level*, a configuration specifies the model, view and controller components that make up a module class. In addition, if the module is composite then the configuration also specifies the module's containment tree. At the *component level*, the configuration specifies properties of each of the three components. First, the **model configuration** specifies properties of the domain class (the model). Second, the **view configuration** specifies such properties as the view's layout and the label, style and dimension of **view fields**. There are two types of view field: subview and data field. A **subview** is a container containing other view fields. A subview presents the view of a descendant module. A **data field**, on the other hand, is a non-composite component, which does not contain other view fields.

Third, the **controller configuration** specifies behavioural properties of the controller through a set of basic actions that are performed in response to the user commands. In particular, the controller configuration specifies the default action that is executed when the view is first displayed. Further, it specifies the policy for an open action, which is executed when the user performs the data openning command. This action connects to the object store of the domain class and prepares the view ready for user to start manipulating the stored objects. Other controller actions are described in [17].

Example 2. In order to configure ModuleEnrolmentMgmt of the CourseMan domain, whose view is shown in Fig. 3, we first need to specify three components: (model, view, controller) = (EnrolmentMgmt, View⟨EnrolmentMgmt⟩, Controller⟨EnrolmentMgmt⟩). In addition, since this module is composite we need to specify a containment tree which details the containment scopes of the modules of the following three domain classes that are associated to EnrolmentMgmt: SClass, Student and HelpRequest.

As for the three component configurations, the *model configuration* contains the model property specified above. The *controller configuration* in this case does not require any specific customisation. The *view configuration* specifies that the overall layout is a tab layout. In addition, it contains labels for the three view fields that render three associative fields that realise EnrolmentMgmt's ends of the associations to the other three domain classes. The labels are used to name the three tabs shown in the figure. Each view field is a subview that displays the view of one child module. For instance, the subview shown in the figure presents the view field View⟨EnrolmentMgmt⟩.students and is also the view of a child module that is typed ModuleStudent. This subview contains four view fields that render four attributes of Student (id, name, helpRequested and modules) that are listed in ModuleStudent's containment scope. These four view fields are data fields.

2.2. Designing the domain model in DCSL

In this paper, we consider **domain model** as a program written in a high-level OOPL, whose meta-model consists in six meta-concepts: Class, Field, Method, Parameter, Annotation and Property. We developed this meta-model in [11] to represent the core structure of two popular OOPLs: Java [18] and C# [19]. The first four meta-concepts of the meta-model are called *non-annotation meta-concepts*. As for the other two meta-concepts, Annotation borrows its name from Java. The equivalent meta-concept in C# is *attribute*. Meta-concept Property generalises annotation element in Java and attribute parameter in C#. A property's type can be an annotation type (or an array thereof). This rule helps reduce design redundancy through annotation reuse. Notationwise, we denote by $A(p_1, \ldots, p_m)$ an annotation A that consists of properties $p_1, \ldots, p_m$. An annotation element e: A is created when annotation A is applied to specify a non-annotation element t. We call this **annotation assignment** and write: $e = A(t)$.

To express the domain models, we defined an aDSL named

Domain class specification language (DCSL) [11]. 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. Finally, meta-concept **Domain Method** is composed of Method with a certain behaviour. The essential behaviours are represented by an annotation named DOpt and (optionally) annotation AttrRef. The latter references the domain field that is the primary subject of a method's behaviour.

For example, Fig. 2, which is adapted from our previous work [11], shows a partial CourseMan's domain model expressed in DCSL. Class Enrolment realises the many-many association between Student and CourseModule (shown in Fig. 1). 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). Domain field Student.enrolments is an associative field, which is assigned with a DAssoc element. This element specifies the Student 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

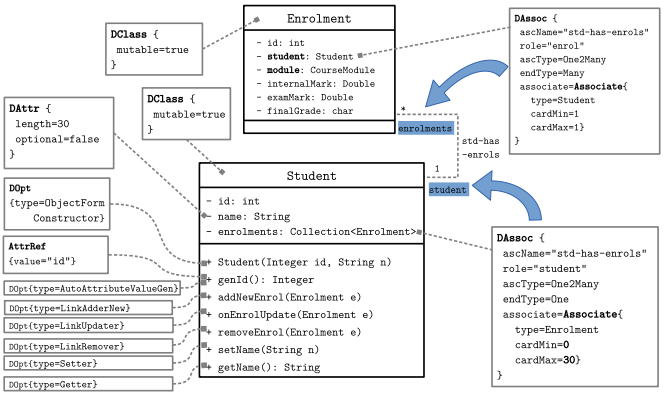

Fig. 2. A partial CourseMan domain model expressed in DCSL.

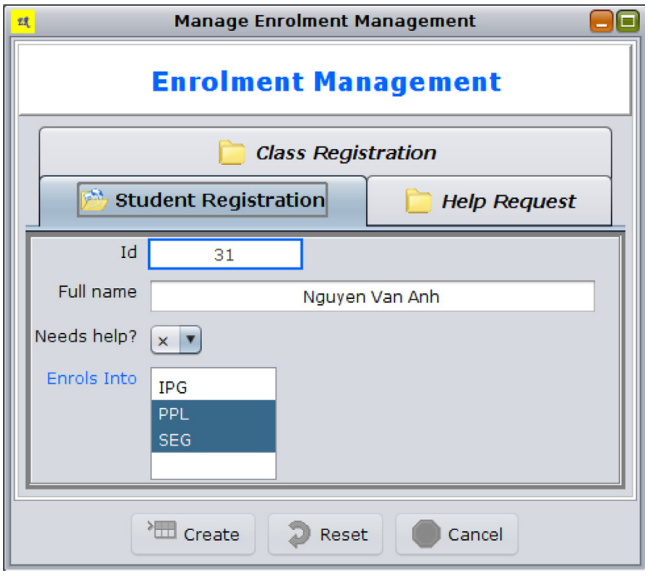

Fig. 3. A GUI with two views: the container view belongs to a `ModuleEnrolmentMgmt` object, the inner view belongs to a descendant `ModuleStudent` object.

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. For instance, method `genId`, whose behaviour type is `AutoAttributeValueGen`, is assigned with an `AttrRef` element, which references the domain field `Student.id`. This means that `genId` is the method that automatically generates values for `Student.id`.

3. Method overview

In order to automatically construct software modules directly from the domain model using aDSL, we need to tackle two key questions: (Q_1) how to define an aDSL to express the MCCs of the modules for automatic generation purpose? and (Q_2) how to enhance productivity further by automatically generating the MCC? To answer these questions, we propose to refine the aDSL-based software development method for DDD that we defined in a previous work [11] with the support for software module development. In the original method [11], domain modelling and software generation are intertwined and performed iteratively through a sequence of three main steps: (1) create/update domain model, (2) generate software and (3) review domain model.

In this paper, as illustrated in Fig. 4, we refine the "generate software" step (step 2) of the original method to provide details for how to generate the software modules that make up a software. The arrowed lines in the figure depict the steps, while the labelled rectangles at the source and target of each line depict a step's input and output. The refinement involves introducing the MCC model, which conforms to our proposed aDSL named MCCL, and five smaller steps (steps 1–4 and 6) that operate on this model. Note that both the initial and functional MCC models conform to MCCL. Steps 3–6 are iteratively performed to construct both the domain and MCC models, until these models are determined by the domain experts to satisfy the domain requirements. In each iteration, a GUI-based software is automatically generated (in step 3) from the MCC model. This software faithfully presents the models (a property known as model reflectivity [11]) and enables the interactive and collaborative work between the domain experts and the technical team. We briefly summarise below the six steps of the refined method:

1. generate the MCCs that make up the initial MCC model from an initial domain model. This step is performed by a key MCC management function named MCCGen.
2. customise the initial MCC model to produce the first functional MCC model.
3. generate a software prototype from the MCC model (performed by JDomainApp [17]).
4. update the MCC model with module-specific changes (changes that concern the non-functional requirements).
5. update the domain model with changes to the functional requirements (see [11]).
6. update the MCC model (if necessary) as the result of the domain model update performed in the previous step. This step is performed by other MCC management functions (see [20]).

Note that steps 1 and 6 aim to answer Q_2. We discuss step 1 in this paper (see Section 6). The two steps 2 and 4 aim to answer Q_1 and are performed using MCCL (see Section 5). Other steps reference the related works, in which they are discussed. We will show that our refined method has the benefit of producing, in the spirit of DDD and with a high and quantifiable degree of automation, the software modules that make up a software. Further, the modules are customisable directly in the target OOPL through the MCCs.

4. Extending MOSA to support MCC model

In order to formulate a DSL for module configuration, we need to make explicit the two notions MCC and MCC model and extend MOSA to formally support them. The extension consists in three enhancements to the module configuration method discussed in Section 2.1. These enhancements help make the method more suitable for practical use and amenable to generative development. Specifically, we present in Sections 4.1 and 4.2 two optimisations of and in Section 4.3 one improvement to the method. To ease discussion, we will treat **module** as being synonymous to module class and **module objects** as objects of this class.

4.1. Master module configuration

Although it is suggested in [12] that it is possible to create more than one module configurations from one domain class, to ease maintenance we propose that one 'master' module configuration is defined from each domain class. This configuration is used as the base for defining the configurations of all the descendant modules of the same class.

More precisely, we assert that there is *a one-one correspondence between a domain class and the master module configuration* that uses that class as the model. We call this module configuration (module class) the **owner module configuration** (owner module class, *resp.*). This assertion helps simplify the containment tree definition in that a descendant node can simply refer to the domain class of the owner module (as opposed to referring to the next module object id [12]). The descendant module object is implicitly the next module object of the descendant module class.

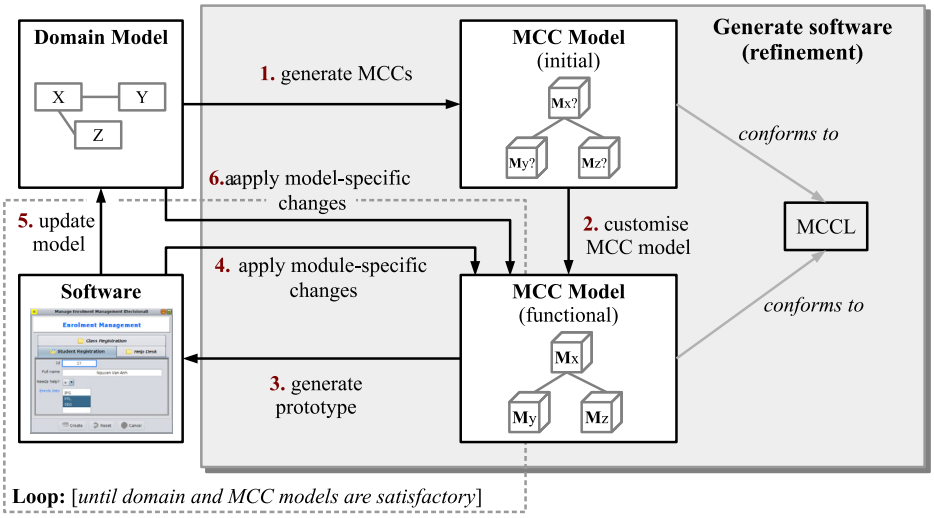

Fig. 4. Method overview: a refinement of the "generate software" step of our aDSL-based software development method (presented in [11]).

With this assertion, we give below the high-level definitions of MCC (based on the domain class that it owns) and MCC model. We will give more precise definitions of these terms later in Section 5.

Definition 1. An **MCC** is a set of module configurations that own the same domain class and that share the same settings for the model, view and controller properties. □

Definition 2. An **MCC model** *w.r.t* a domain model is a set of MCCs, each of which owns a domain class in the model. Thus, an MCC model defines the configurations for a sub-set of modules that make up a software. □

4.2. The 'Configured' containment tree

An important property of our module configuration method is that it does not require the designer to specify the whole containment tree of a composite module. She only needs to focus on the edges, whose associated containments require customisation. Further, the configuration of each of these edges only need to contain the customised values for the concerned containment. The main reason for this is that the entire containment tree of a composite module can be automatically generated, using the procedure described in [12]. The input of this procedure is the sub-model of the domain model that contains the domain classes that are associated (directly and indirectly) to the composite module's domain class. The generated containment tree contains the default configuration values for all the nodes and edges.

The above property allows us to use the notion of **configured containment tree** when working in the module configuration context. To ease discussion in the remainder of the paper (except for in the next section) and when it is not necessary to make a distinction with the **actual containment tree**, we will use the term "containment tree" to mean the configured one.

4.3. Customising descendant module configuration

The module configuration proposed in [12] supports not only module class but also module objects of this class that are descendants in the containment tree of some composite module. However, the current configuration of these module objects includes only a customised containment scope. To achieve a complete design freedom for the descendant modules, we propose to extend the configuration to cover all three MVC components. In practice, for example, it may be necessary to make the model uneditable, to change the layout of the subview of a module object, or to alter the display type of a particular view field of the subview. We propose three essential properties that this customised configuration should possess. We derive these properties from those of DSL. Usability is necessary to ease learning and using [21], while modularity and reusability are two important properties of embedded DSL [22]:

- *usability*: the custom configuration is defined in the same manner as the standard configuration (i.e. using annotation).
- *modularity*: the custom configuration is captured by a separate sub-model, which is connected to the descendant module's configuration.
- *reusability*: the custom configuration needs to reuse as many as possible the meta-concepts that are being used in other parts of the configuration to specify the same properties.

5. MCCL: an ADSL for expressing MCC model

We present in this section the specification of our proposed **module configuration class language** (**MCCL**). MCCL extends and formalises the module configuration template that was first introduced in [12]. We first give an overview of the language specification method. After that we specify the language syntax using UML/OCL.

5.1. Specification method

We apply the **meta-modelling approach** [23,24] to define MCCL. Our aim is to construct the ASM in the annotation-based form, expressed in the OOPL's meta-model. To construct the ASM requires three steps. In the first step, we construct a conceptual model (CM) of the domain as a UML/OCL class model. 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. We strive to produce a compact CM_T so that this would result in a compact ASM (i.e. consisting in a small set of annotations). From the practical standpoint, such an ASM is desirable because it would require less effort from the language user to construct an MCC. In the third step, we then transform CM_T into the actual ASM, which takes an annotation-based form specified in the OOPL's meta-model. In this form, the configuration-related meta-concepts are represented by annotations. Further, we use Class and Field as the basis to structure the annotations. The module-related annotations are attached to Class, while the view field annotations are attached to Field. The resulted Class-based structure literally becomes an MCC.

MCC enables us to treat module configurations as first-class objects, which are constructed along side the domain model. They also help ease

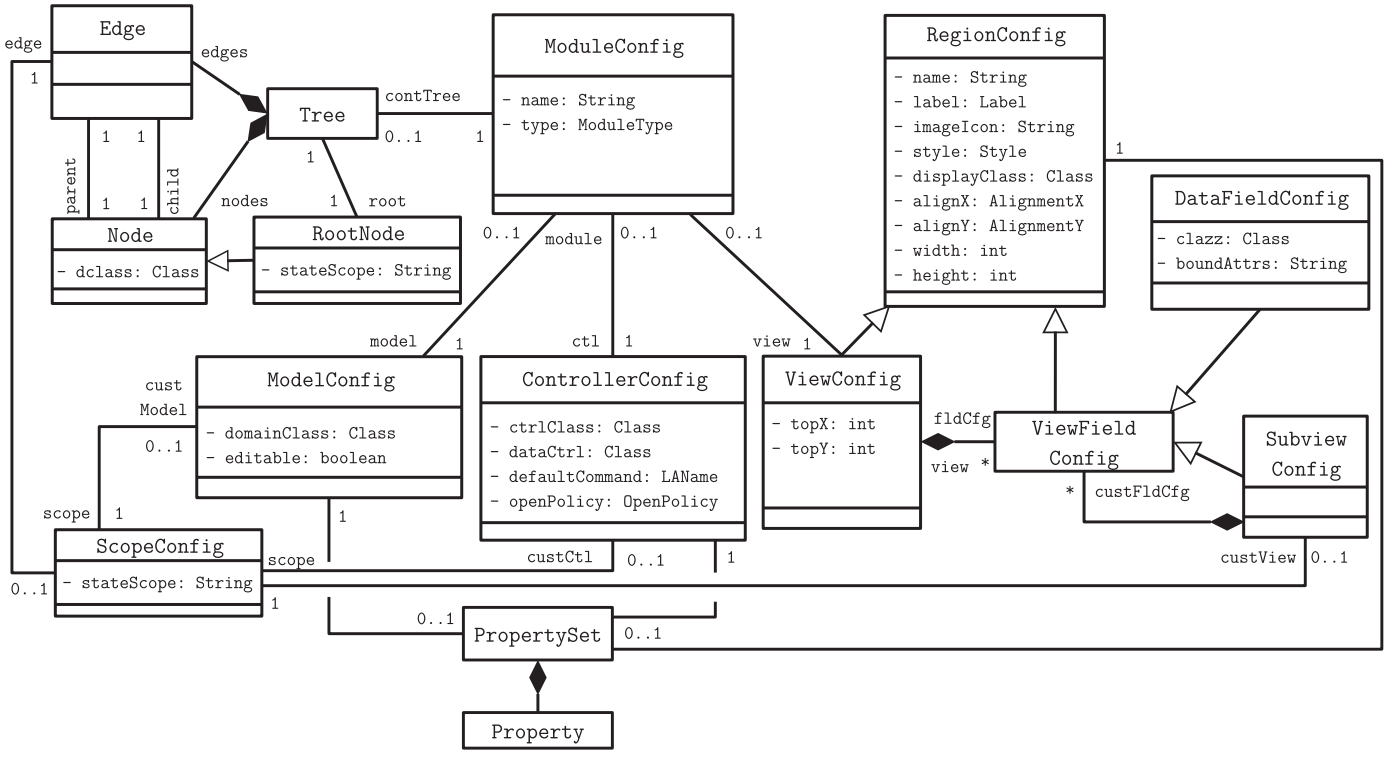

Fig. 5. The conceptual model of MCCL.

the development of the overall software. In particular, the MCCs are reusable, on behalf of the target (generated) modules, to create different software. To ease discussion in the remainder of this chapter, we will (unless explicitly stated otherwise) consider MCC as being synonymous with module class. In particular, we will say that module objects are objects of MCC. Further, because both ASM and CSM are represented as class models and to ease notation we will refer to meta-concepts that appear in a meta-model as classes.

5.2. Conceptual model

Fig. 5 shows the UML class diagram of the conceptual model (CM) of the MCCL's domain. Class `ModuleConfig` represents module configuration, while classes `ModelConfig`, `ViewConfig` and `ControllerConfig` represent the three component configurations (*resp.*). The module containment tree is represented by the association from `ModuleConfig` to `Tree`. The structure consisting of `Tree` and three other classes (`Node`, `RootNode` and `Edge`) define a labelled, binary and rooted tree. Class `Node` has one attribute, named `dclass`, which specifies the domain class of the corresponding module. `RootNode` is a subtype of `Node` which contains an additional attribute named `stateScope`. This is used to specify the state scope of the corresponding module. Class `Edge` has two associations to `Node`, which clearly specify that its two `Nodes` play the roles of parent and child.

Class `ViewConfig`, in particular, is composed from one or more `ViewFieldConfigs` and, if it is the view of a composite module, then also from one or more `SubviewConfigs`. `DataFieldConfig` and `SubviewConfig` are sub-types of `ViewFieldConfig`, which represent the configurations of data field and subview, respectively. Class `RegionConfig` is a generalisation of both `ViewConfig` and `ViewFieldConfig`. Conceptually, `RegionConfig` represents any GUI region that forms part of a view. This can be a view field, a subview, or an entire view.

The CM contains a sub-model for customising the configuration of descendant module. This sub-model is designed with an aim to satisfy the three design properties discussed in Section 4.3. More specifically, the sub-model consists of four classes (`ScopeConfig`, `ModelConfig`, `ControllerConfig` and `SubviewConfig`) and four associations. The latter three classes are re-used from other parts of the CM. The first three associations are between these classes and `ScopeConfig`. These associations are used to customise the three component configurations. The fourth association is the composition association between `SubviewConfig` and `ViewFieldConfig`. This association is used to customise the configurations of the view fields of the descendant module's view. The entire sub-model is connected to class `Edge` via the association between this class and `ScopeConfig`. This associates every customised configuration to a containment edge.

To ease the introduction of other configuration properties in the future, we use `PropertySet` and `Property`. The former is composed of the latter and is referenced by `ModelConfig`, `ControllerConfig` and `RegionConfig`.

Well-formedness Rules

We use OCL invariant to precisely express the well-formedness rules of the CM. We group the rules by the meta-concepts of the CM to which they are applied. Each rule includes a header comment that describes, in natural language, what the rule means. For the interest of space, the rule definitions are presented in the extended paper [20]. Syntactically, some rules use DCSL to express constraints on certain meta-concepts' attributes. There are two reasons for this. First, it is possible to apply DCSL to the CM in Fig. 5, because this model contains only the elements that are mappable directly to those expressible by DCSL. Specifically, it includes only class, field, one-one and one-many associations, and generalisation. Second, it is more compact and intuitive to write rules in DCSL, where applicable. For instance, the OCL expression `name.Unique` is shorter and easier to grasp than the equivalent, but lengthy, OCL rule describing the fact that attribute `ModuleConfig.name` is unique (i.e. its values are distinct among all the `ModuleConfig` objects).

5.3. Abstract syntax

In order to construct a compact ASM from the CM that takes the annotation-based form (suitable for being embedded into a host OOPL)

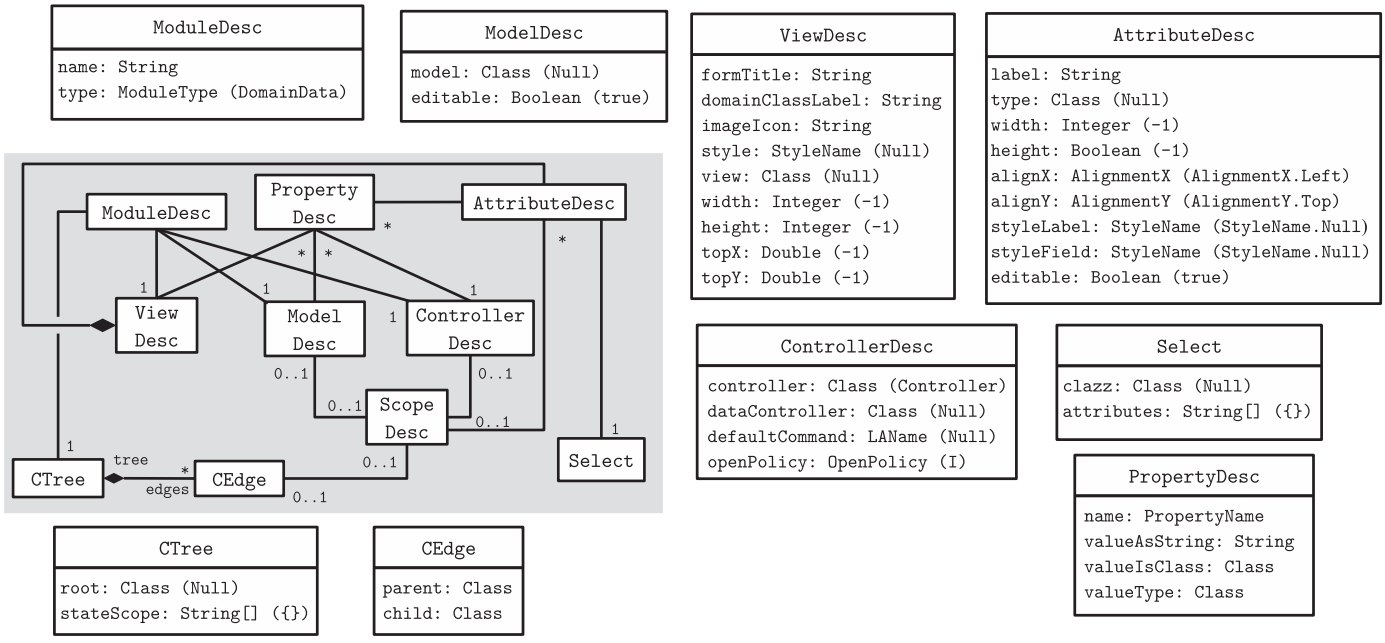

Fig. 6. (A-shaded area) The transformed CM (CM_T); (B-remainder) Detailed design of the key classes of CM_T.

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. After that, we explain the ASM.

5.3.1. CM_T: a compact and annotation-friendly model

Fig. 6 (A) shows the UML class model of CM_T. The detailed design of the key classes are shown in Fig. 6(B). We present the well-formedness rules of CM_T and the mapping from CM to CM_T in the extended paper [20]. To ease our 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.

In Fig. 6(A), class `ModuleDesc` represents the overall module configuration, while class `AttributeDesc` represents the view field configuration. Class `ModuleDesc` has associations to the following three classes that specify the three component configurations: `ModelDesc`, `ViewDesc` and `ControllerDesc`. In addition, class `ModuleDesc` has an association to `CTree`, which specifies the containment tree configuration.

The tree structure `Tree-Node-RootNode-Edge` of the original CM is replaced by the structure `CTree-CEdge` in CM_T. This new tree representation is more compact and fits naturally with the idea of the configured containment tree (see Section 4.2). That is, instead of having to specify the entire tree the designer only configures the containment edges that need to be customised. More specifically, class `CEdge` has two attributes, named `parent` and `child`, that specify the `dclasses` of the parent and child `Nodes` of an `Edge` in the CM. Class `CEdge` also has an association to a class named `ScopeDesc`, which is mapped directly to the class named `ScopeConfig` of the CM.

5.3.2. 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 the actual ASM of the language that is "embedded" in OOPL. This ASM is constructed from the following four OOPL meta-concepts: Class, Field, Annotation and Property. Fig. 7 shows the UML class model of the ASM. In this, the classes in CM_T are transformed into annotations of the same name. Each domain field is transformed into an annotation property. The annotations are depicted in the figure as grey-coloured

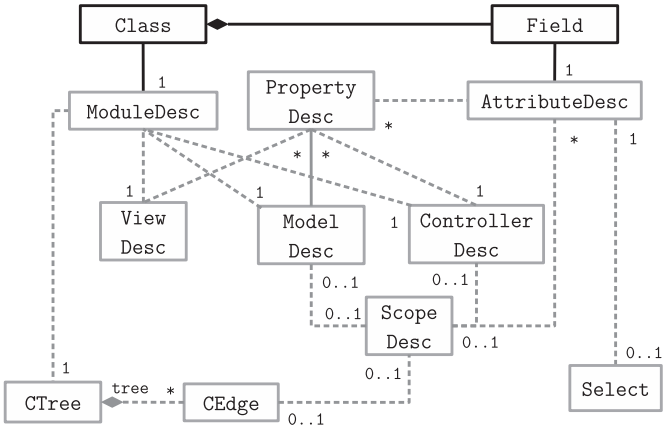

Fig. 7. The annotation-based ASM of MCCL.

boxes. The grey dashed association lines that connect the annotations are used merely as a visual aid to relate the overall ASM's structure to the corresponding associations in CM_T.

A key structural difference between ASM and CM_T is the addition of two annotation attachments: `ModuleDesc` to `Class` and `AttributeDesc` to `Field`. These are represented in Fig. 7 by solid lines connecting the corresponding class boxes. A `ModuleDesc` attachment defines an MCC because it describes the instantiation of a `ModuleDesc` object together with objects of the annotations that are referenced directly and indirectly by `ModuleDesc`. The association between `Class` and `Field` realises the composite association between `ViewDesc` and `AttributeDesc`. The reason is that there is a one-one association between `ModuleDesc` and `ViewDesc` and, thus, associating `ModuleDesc` to `Class` leads to `ViewDesc` being associated to it. The `Fields` that make up a `Class` together with the `AttributeDescs` of these fields constitute the view field configurations of the `ViewDesc` attached to that `Class`.

Together, the above features lead us to the following (more precise) definition of MCC and view field.

Definition 3. An **MCC** is a class assigned with a suitable `ModuleDesc`, that defines the configuration of a module class owning a domain class

```
@ModuleDesc(name="ModuleStudent",
modelDesc=@ModelDesc(model=Student.class),
viewDesc=@ViewDesc(formTitle="Manage Students",imageIcon="student.jpg",
  view=View.class, parentMenu=RegionName.Tools, topX=0.5,topY=0.0),
controllerDesc=@ControllerDesc(controller=Controller.class,
  openPolicy=OpenPolicy.I_C),
containmentTree=@CTree(root=Student.class,
  stateScope={"id", "name", "modules"}))
public class ModuleStudent {
  @AttributeDesc(label="Student")
  private String title;

  @AttributeDesc(label="Id",type=JTextField.class,alignX=AlignmentX.Center)
  private int id;

  @AttributeDesc(label="Full name",type=JTextField.class)
  private String name;

  @AttributeDesc(label="Needs help?",type=JBooleanField.class)
  private boolean helpRequested;

  @AttributeDesc(label="Enrols Into",type=JListField.class
  ,ref=@Select(clazz=CourseModule.class,attributes={"name"}),
  width=100,height=5)
  private Set<CourseModule> modules;
}
```

Listing 1. The MCC of `ModuleStudent`.

in the domain model. Further, the MCC's body consists of a set of fields that are assigned with suitable `AttributeDescs`. These fields realise the view fields. Exactly one of these fields, called the **title data field**, has its `AttributeDesc` defines the configuration of the title of the module's view. Other fields have the same declarations as the domain fields of the domain class and have their `AttributeDescs` define the view-specific configuration of these domain fields.

We say that a view field **reflects** the corresponding domain field. To ease discussion, we will say that the MCC owns the module's domain class. □

Now, a set of MCCs that provide the configurations for the owner modules of a subset of domain classes in a domain model forms an MCC model (see Def. 2). This model conforms to MCCL.

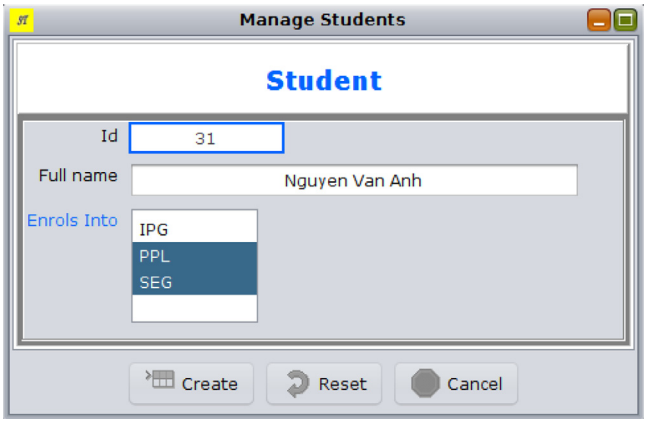

Fig. 8. The view of a stand-alone `ModuleStudent`, configured by Listing 1.

5.4. Concrete syntax

Because MCCL is embedded into OOPL, it is natural to consider the OOPL's textual syntax as the concrete syntax of MCCL. From the perspective of concrete syntax meta-modelling approach [24], the CSM of such textual syntax is derived from that of OOPL. Further, the core structure of the CSM model is mapped to the ASM. In addition to this core structure, the CSM contains meta-concepts that describe the structure of the BNF grammar rules. The textual syntaxes of Java and C# are both described using this grammar.

In this paper, we will adopt the Java textual syntax as the concrete syntax of MCCL. A key benefit of the Java syntax (as discussed in Section 2.2) is that its annotation supports annotation-typed property, which enables annotation reuse and a structural composition scheme. We illustrate this syntax using the MCCs of two CourseMan software modules that we discussed in Examples 1 and 2. The first MCC is `ModuleStudent` and the second is `ModuleEnrolmentMgmt`. Due to space restriction, we present `ModuleEnrolmentMgmt` in A.2

Example 3. (`ModuleStudent`) Let us consider the MCC of `ModuleStudent` shown in Listing 1. The resulted module's view is presented in Fig. 8.

The MCC's header is assigned with a `ModuleDesc` element, which defines the module configuration: the module's name is "`ModuleStudent`", the referenced `ModelDesc` specifies the domain class `Student`, the `ViewDesc` specifies the view (e.g. title = "`Manage Students`"), the `ControllerDesc` specifies the controller class to `Controller` and the object opening policy to `I_C`.

The `ViewDesc` element in Listing 1 is specified with a property named `parentMenu`. This property is supported by JDomainApp, which creates a menu item in the software GUI for the user to access the module. Since this property is not intrinsic to MCCL, it will be omitted from discussion in the remainder of this paper.

The referenced `CTree` of `ModuleDesc` specifies that the descendent module owning `Student` has the containment scope containing only three attributes (`id`, `name`, `modules`). It excludes the domain field

Student.helpRequested. In the next example, we will explain how ModuleStudent is reused to create another ModuleStudent object that is a descendant of ModuleEnrolmentMgmt. The subview of this object is customised to contain data fields for all four Student's domain fields.

The MCC's body contains five view fields, each of which is assigned with an AttributeDesc element. All five view fields are data fields. The first configuration specifies the data field that presents the view's title text. The remaining four configurations specify the data fields that are labelled (via property AttributeDesc.label) "Id", "Full Name", "Needs help?" and "Enrols Into". These reflect the following four domain fields of Student: id, name, helpRequested and modules (*resp.*). The data fields are configured (using property AttributeDesc.type) with a specific display class type. Specifically, the display class types of the four data fields are JTextField, JTextField, JBooleanField and JListField (*resp.*). If omitted, the display class type of a data field will be derived automatically from the domain field's data type. For instance, property AttributeDesc.type can be omitted from the configuration of the first three data fields.

The data field modules is configured with AttributeDesc.ref.attributes = {}name″ }. This means that this data field is to display values of the domain field CourseModule.name. Indeed, Fig. 8 shows how the data field (which is labelled "Enrols Into") displays the names of three CourseModules (}}IPG″, }}PPL″ and }}SEG″) that are currently in the system. These values are easier for the user to comprehend than those of the identifier attribute CourseModule.code. □

6. Automatically generating MCC from the domain class

Because an MCC has a structurally close relationship to its domain class, it needs to be managed effectively in order to ensure that this structural relationship is preserved when the domain class as well as the MCC evolve. Further, in order to reduce the effort needed to construct the MCCs for a particular software, the MCC management mechanism should incorporate some form of automation. In this section, we discuss a key MCC management function, named MCCGEN, that is responsible for automatically generating an MCC from the domain class. This function forms part of the MCCL's toolkit that is provided for the language users. It is worth mentioning that, for completeness, we have defined in the extended paper [20] other MCC management functions, including update-on-change and update-on-deletion. In the interest of space, however, we will not discuss these functions in this paper.

We present in Algorithm 1 the algorithm for function MCCGEN. This function takes as input a domain class (c) and generates as output the 'default' owner MCC (m) of that class. By 'default' we mean the generated MCC contains the default values for all the essential annotation properties. To ease comprehension, we insert comments at the key locations to help explain the algorithm.

Specifically, Algorithm 1 consists of three main steps, which are labelled "STEP 1"–"STEP 3" in three comments. In step 1, the MCC header is created. In particular, the class name includes the prefix 'Module' combined with the domain class's name. In step 2, three configuration component elements of the MCC (d_o, d_v, d_c) are created first, then the ModuleDesc element (d_m) is created from them.

In step 3, the view fields are created in the MCC's body. The first view field (f_d) is the title field. Other view fields are created for the domain fields (set c_F) of c. Since this is a shared procedure, we define it as a function named

AddViewFields. This function, which is displayed at the bottom of Algorithm 1, basically consists in a for loop that iterates over the input domain field set (F) to extract the name and data type of each field (f) and use these to create a corresponding view field (f_d) in the output class (MCC m). The AttributeDesc of each view field has its label property set to name of the corresponding data field (label=n_f).

The parts labelled ❶–❹ in Algorithm 1 contain properties which may need to be customised to suite the domain requirements. This customisation concerns module-specific changes. More specifically, part ❶ includes changes to view-related properties, such as title, label and icon, and other properties discussed in Section 5.2. For example, if the module does not have a view then property ViewDesc.view needs to be changed from View to Null. Part ❷ includes changes to the overall module configuration. For instance, if the module is composite then property ModuleDesc.containmentTree would be set to a suitable CTree. Parts ❸ and ❹ entail changes (via AttributeDesc) to the view configuration of the data fields representing the title and the domain fields of the input domain class.

Algorithm 1 MCCGEN.

Input: c: Domain Class
Output: m: MCC

// STEP 1: create m's header
1: $n_c \Leftarrow c$.name, n_m = "Module" + n_c
2: $m \Leftarrow$ Class(visibility="public",name=n_m)
// STEP 2: create m's ModuleDesc
3: $d_o \Leftarrow$ ModelDesc(model=c)
4: $d_v \Leftarrow$ ViewDesc(formTitle="Form: "+n_c,domainClassLabel=n_c, imageIcon=n_c+".png",view=View)// ❶
5: $d_c \Leftarrow$ ControllerDesc()
6: $d_m \Leftarrow$ModuleDesc(m):modelDesc=d_o,viewDesc=d_v, controllerDesc=d_c // ❷
// STEP 3: create m's view fields (i.e.~view field configs)
7: $f_d \Leftarrow$ Field(class=m,visibility="private",name="title", type=String)// title
8: create AttributeDesc(f_d):label=n_c// ❸
9: $c_F \Leftarrow \{f \mid f$: Domain Field, $f \in c$.fields$\}$ // c's domain fields
10: AddViewFields(m, c_F) // create a view field to reflect each c's domain field
11: **return** m

12: Function **AddViewFields**(Class m, Set⟨Field⟩ F):
13: **for all** f in F **do**
14: $n_f \Leftarrow f$.name, $t_f \Leftarrow f$.type
15: $f_d \Leftarrow$ Field(class=m,visibility="private",name=n_f, type=t_f)
16: create AttributeDesc(f_d):label=n_f// ❹

Example 4. (Generating ModuleEnrolmentMgmt) Listing 2 shows the owner MCC ModuleEnrolmentMgmt of EnrolmentMgmt, which is generated by MCCGEN. The model configuration specifies that the domain class is EnrolmentMgmt. In the view configuration, the values of the ViewDesc's properties are set to the default values. For instance, formTitle and imageIcon are constructed directly from the domain class's name. The designer would customise these values to suite the domain. In particular, formTitle would be changed to a more user-friendly label, such as

}}Manage Enrolments″. The default controller configuration requires no property value specification. The containment tree is the (actual) automatically-generated tree. The designer would customise this tree for some descendant modules of interest. An example of such customisation is described in A.2.

The MCC's body contains the declarations of five view fields. The labels of these fields take the default values, which are the same as the field names. These labels would be customised to contain user-friendly texts. An example configuration of these fields is shown in Listing 4 of Appendix A. □

Analysis of MCCGEN

We analyse the correctness and performance of function MCCGEN in order to be confident that it works not only as expected but effectively in practice. We state the theorems concerning these two properties of

```
@ModuleDesc(name = "ModuleEnrolmentMgmt",
modelDesc = @ModelDesc(model = EnrolmentMgmt.class),
viewDesc = @ViewDesc(view = View.class, formTitle = "Form: EnrolmentMgmt",
  imageIcon = "EnrolmentMgmt.png", domainClassLabel = "EnrolmentMgmt"),
controllerDesc = @ControllerDesc())
public class ModuleEnrolmentMgmt {
  @AttributeDesc(label = "title")
  private String title;

  @AttributeDesc(label = "id")
  private int id;
  @AttributeDesc(label = "students")
  private Set<Student> students;
  @AttributeDesc(label = "helpDesks")
  private List<HelpRequest> helpDesks;
  @AttributeDesc(label = "sclassRegists")
  private Collection<SClassRegistration> sclassRegists;
}
```

Listing 2. The 'default' `ModuleEnrolmentMgmt` produced by MCCGEN.

the function and provide proofs for them. Details of the theorems and proofs are presented in A.3.

7. Tool support

Tool support for our proposed method consists of MCCL and function MCCGEN. We implemented these components as an update to the software tool that we developed in the previous work [11]. This tool includes DCSL and its associated components. The software tool was implemented in a Java-based software framework [17], named JDOMAINAPP. Our implementation, which is accessible online [25], uses a third-party parser named JavaParser [26] to parse the Java code model into a syntax tree for manipulation. We have used our software tool to develop real-world software, one of which is a case study that we will report in Section 8.2.

A developer can use our software tool in one of three ways: (*i*) execute the components directly (as described in the online manual [25]), (*ii*) execute the components using an Eclipse plugin or (*iii*) treat the components as libraries for importing into an existing code base. A basic development procedure follows the method flow presented in Fig. 4, with an additional note that the domain model is constructed using DCSL and its associated tool components.

The Eclipse plugin has recently been developed and is reported in detail in [27]. As far as this paper is concerned, the plugin includes components for automating the construction of the domain class using DCSL and a component for the MCCGEN function. Other useful components concern the automatic generation of a software from an MCC model. Fig. 9 shows a screenshot of the Eclipse plug-in action. At the bottom of the "Source" sub-menu of the context menu is a list of functions provided by the plugin. The code editor presents the default MCC of `ModuleEnrolmentMgmt`, which we showed in Listing 2 of Example 4. This MCC was generated by function MCCGEN, which was invoked on the `EnrolmentMgmt` class through the "Generate MCC" menu item.

8. Evaluation

Our evaluation seeks to answer the following core questions concerning our method:

1. *How well does our method perform against a given domain model?*
   Theorems 1 and 2, which concern the analysis of function MCCGEN (see Section 6), together show that our method performs well against the input domain model. It correctly generates an MCC from a domain model and does so with a linear time complexity (*w.r.t* the size of the input domain class).
2. *What is the module generativity extent of our method?*
   We answer this question by defining a formal evaluation framework for module generativity. We present this framework in Section 8.1.
3. *Is our method applicable to developing real-world software?*
   We answer this question with a representative case study that will be presented in Section 8.2.

8.1. An evaluation framework for module generativity

Our objective is to evaluate the extent to which MCCL helps automatically generate software modules. To achieve this, we define a framework for developer-users of MCCL to precisely quantify module generativity for the application domains. In the framework, we will identify the general module patterns and discuss how module generativity can be measured for each of them.

8.1.1. Method

We first observe that the module generativity procedure consists of two steps: (*i*) generate the MCC of the module and (*ii*) generate the module from the MCC. The first step is performed semi-automatically by function MCCGEN. The second step is performed semi-automatically by the module interpreter of JDOMAINAPP. We measure module generativity by taking a *weighted average* of the generativity values of these steps.

Another observation that we make is that the parts that need to be manually written by the developers in both steps should and can be identified through the module configuration captured by the MCC. This is useful because then we can measure the generativity value of each step by the efforts taken to compose those parts. We will call the parts generally by the name *customised elements*. The customised elements in the first step are called *customised configuration elements*, while those in the second step are called *customised components*. To facilitate the measurement, we further classify each type of elements by the component type: *view elements* and *controller elements*. We omit the model component (the domain class) of each module from measurement because it is developed before hand.

Table 1 describes a classification of

module patterns (**MPs**), based on valid customisation scenarios. A valid customisation scenario correctly describes a combination of element types that need or need not be customised. In the table, the for-

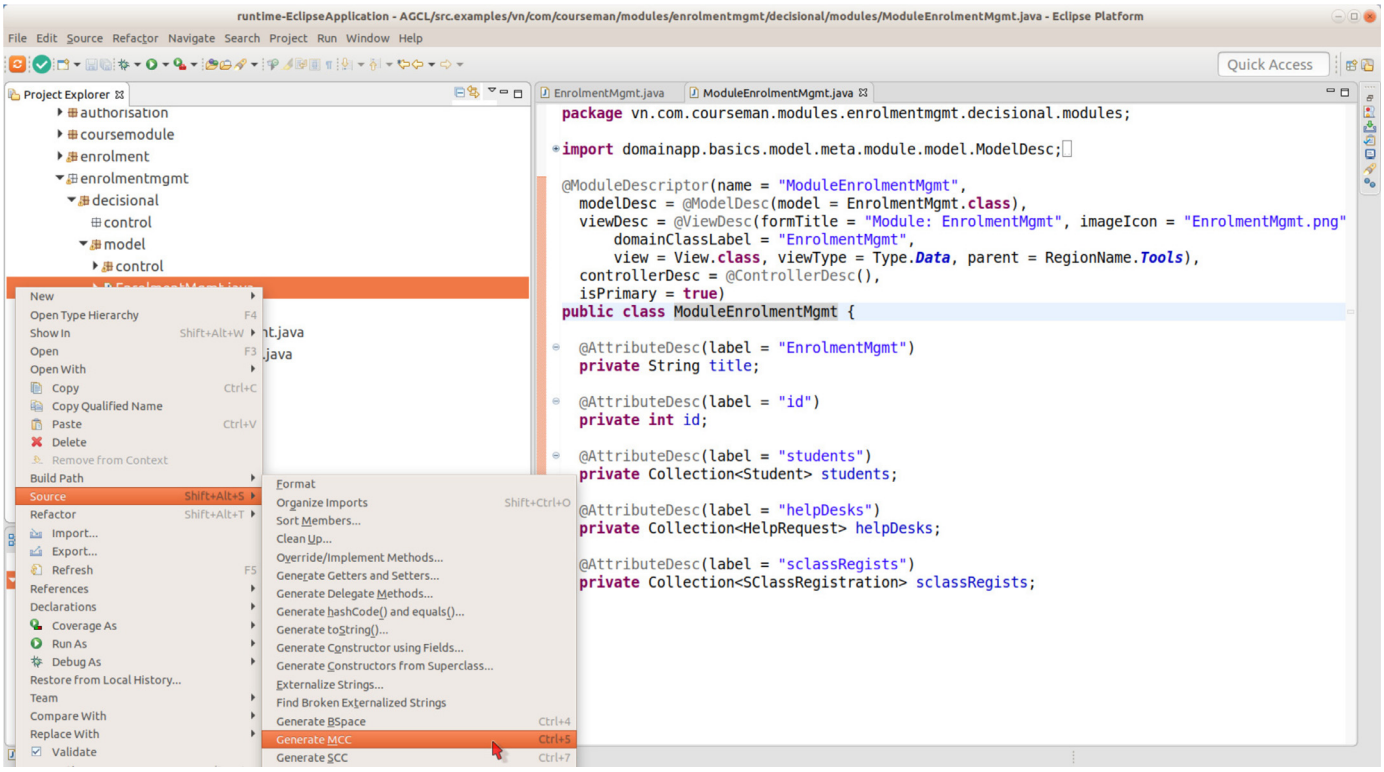

Fig. 9. Screenshot of the Eclipse plugin of our method.

Table 1
Module pattern classification.

MPs	View			Controller		
	Configuration		Code	Configuration		Code
	Def	Cust	Cust	Def	Cust	Cust
MP_1	√			√		
MP_2	√				√	(√)
MP_3		√	(√)	√		
MP_4		√	(√)		√	(√)

mer case is abbreviated as "*Cust*" (customised), while the latter case is abbreviated as "*Def*" (default). Conceptually, an MP represents a set of module objects of one or more module classes that result from applying a particular customisation pattern on these classes. Thus, the four MPs provide a complete partition on the set of module objects of the module classes constructible by our method.

Note the followings about our MP classification. First, the module objects that make up a containment tree can belong to different MPs. For instance, a composite module may belong to MP_1, while its descendant modules belong to other MPs. Second, we do not need to consider the *Def* case for code because this is implied if no code customisation is performed. Third, a customised configuration (of either view or controller) *may* require creating customised component(s). In the table, we show this causal relationship by writing (√) in the *Cust-Code* column whenever we use a √in the *Cust-Configuration* column. Fourth, we systematically define customised components around design patterns that extend the MOSA's functionality. A key benefit of this is code reuse and, hence, improved module generativity for new modules. For controller, a customised component is a pattern-specific module action, which is an action whose behaviour is customised to satisfy a new design requirement. Evans [28] proposed a set of high-level patterns, while we defined in [16] a number of concrete patterns. For view, a customised component is a new view component that improves the usability of the module view. This includes a view field class, a new view layout component and so on. We discuss different view components in [17].

A last observation that we make is that we can construct a shared general formula for the generativity factors of both steps of the module generativity procedure. The basis for this formula is that both steps involve writing some form of code for view and/or controller. Recall that MCC is also a form of code.

Denote by $\mathcal{V}$ and C the amounts of code created for the view and controller components (*resp.*) and by $\mathcal{V}'$ and C' the amounts of customised code that need to be manually written for these same two components (*resp.*). In other words, $\mathcal{V}' + C'$ is the amount of developer's code input. Further, let $\mathcal{W} = \mathcal{V} + C$ be the total amount of code created for the view and controller. Denote by m the generativity factor, then we measure m in (0,1] by the following formula:

$$m = \frac{(\mathcal{V} - \mathcal{V}') + (C - C')}{\mathcal{W}} = 1 - \frac{\mathcal{V}' + C'}{\mathcal{W}} \tag{1}$$

Denote by m_1, m_2 the generativity factors of steps 1 and 2 (*resp.*). We use subscripts 1 and 2 to denote the components of m_1 and m_2, *resp.*; e.g. $\mathcal{W}_1$, $\mathcal{W}_2$, *etc.* We measure the module generativity M (also in (0,1]) by taking a weighted average of m_1 and m_2. We compute two weights α and $1 - \alpha$ based on the relative extent of $\mathcal{W}_1$ and $\mathcal{W}_2$:

$$M = \alpha m_1 + (1 - \alpha) m_2 \quad \left(\text{where: } \alpha = \frac{\mathcal{W}_1}{\mathcal{W}_1 + \mathcal{W}_2}, 1 - \alpha = \frac{\mathcal{W}_2}{\mathcal{W}_1 + \mathcal{W}_2}\right) \tag{2}$$

In Formula 2, the higher the value of M, the higher the module generativity. Our choice of weights means to give higher emphasis to the component (m_1 or m_2) that dominates in the impact on M. In practice, the dominating factor is typically m_2, because $\mathcal{W}_1$ (configuration) is typically much shorter than $\mathcal{W}_2$ (actual code). This leads to module generativity being high or low typically depends on whether or not we need to create a small or high number of customised components to extend the MOSA's core functionality. We will use Formula 2 as the basis to develop more specific formulas of M for the four MPs. We illustrate each formula and the MP(s) in question by a version of the CourseMan software that is constructed by the software tool of our method.

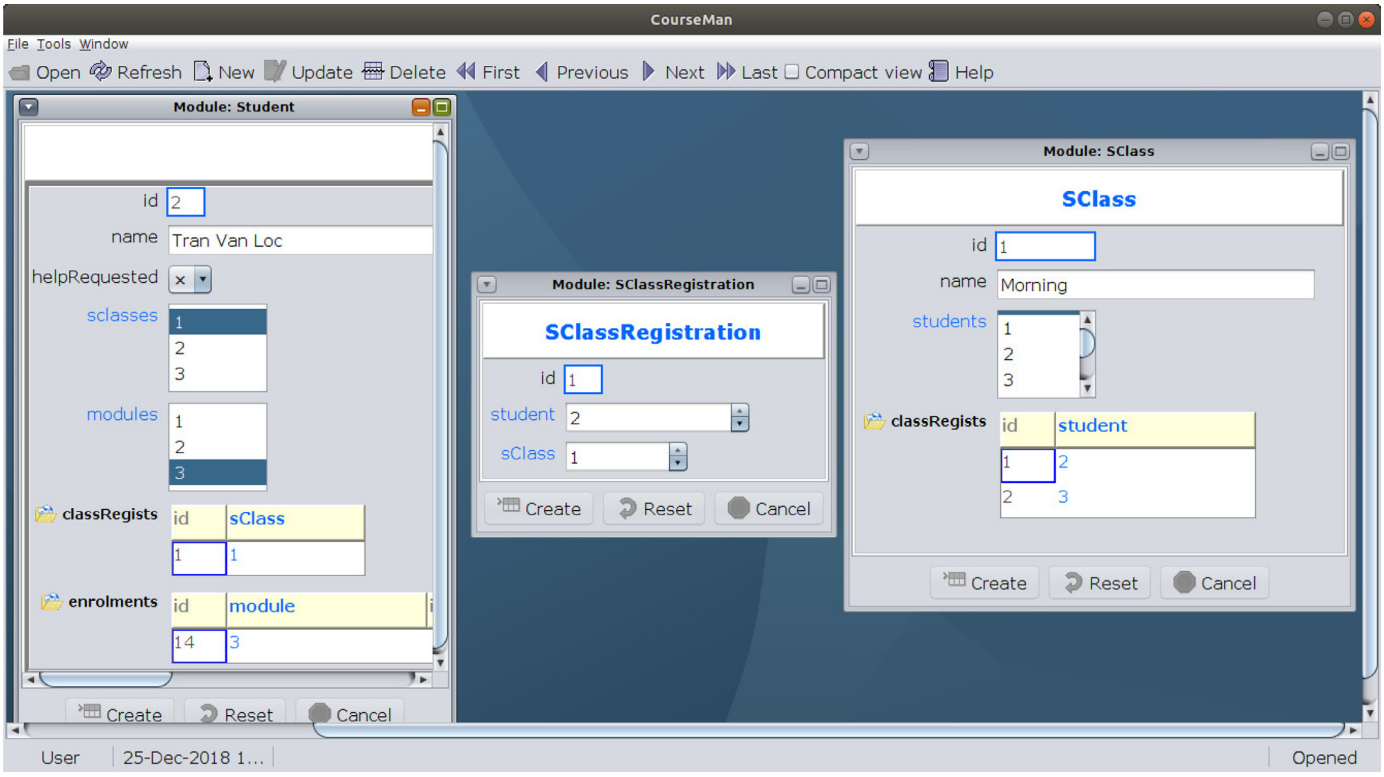

Fig. 10. A partial CourseMan's GUI that is generated by DomainAppTool.

8.1.2. $MP_{1:\text{ total generativity}}$

In this special MP, we achieve 100% generativity, because $\mathcal{V}'_1 = \mathcal{V}'_2 = 0$ and $C'_1 = C'_2 = 0$, which leads to $M = m_1 = m_2 = 1$. A reference software that is constructed only by modules belonging to this MP is implemented by a JDomainApp tool, which we call DomainAppTool. The software generated by this tool has the same GUI as those described in the CourseMan examples in the earlier chapters of this paper, except that every view field (of a module's view) is generated with a default configuration. This default configuration is sufficient for the developer to directly observe and test the domain class design.

DomainAppTool enables the three key stakeholders of the development process to collaboratively and iteratively build a domain model. The designer uses the tool to iteratively work with the domain expert to capture the domain requirements in the model. The programmer participates in the iterations by initially coding the domain model and subsequently updating it, based on changes to the requirements that she has received from the domain expert. The model is processed automatically by the tool to generate the software. In each iteration, the tool automatically generates an interactive software prototype directly from the domain model. This software's GUI shows an *object form* that realises a module's view and a desktop for organising these forms. The object form provides an interface for the user to view and to manipulate the domain objects of the module's domain class. We have been using DomainAppTool to teach in a software engineering course module at the Faculty of IT, Hanoi University[5]. In this course module, students and teachers play the stakeholders' roles.

Example 5. Fig. 10 shows an example CourseMan software that is generated by DomainAppTool. It includes three views: `View⟨Student⟩`, `View⟨SClassRegistration⟩` and `View ⟨SClass⟩`.

`View⟨Student⟩`, in particular, has three list-typed view fields: `View⟨Student⟩.sclasses`, `modules` and `students`. These fields reflect the two many-many associations between `Student` and `SClass` and `CourseModule` (*resp.*). The actual handling of user actions on these fields require customising some module actions. This customisation will be discussed later as part of the fourth MP. For this MP, these view fields are used by the development team to visually review the domain class design. □

8.1.3. MP_2–MP_4

These three MPs involve customising view and/or controller at various extents. We thus first develop a shared formula for all three MPs. After that, we discuss how it is applied to each MP. Denote by W and w the numbers of customised configuration elements of a module and of a view field (*resp.*).

Definition 4. The **configuration customisation factor** of a module, whose view contains s number of view fields, is:

$C = (W + \sum_{i=1}^{s} w_i)$

□

Assuming that the module's containment tree has n number of descendant modules, whose configurations need to be customised. Let C_k be the configuration customisation factor of the k^{th} descendant module. We derive from Formula 1 the following formula for m_1:

$$m_1 = 1 - \frac{C + \sum_{k=1}^{n} C_k}{\mathcal{W}_1} \tag{3}$$

Denote by P and p the numbers of lines of code (LOCs) for the customised components of a module and of a view field (*resp.*). Note that code customisation occurs at both the module level and the view field level.

Definition 5. Given that a module has T number of customised components at the module level, s number of view fields and t_i number of customised components for the i^{th} view field of the module view. The **code customisation factor** of the module is:

$D = \sum_{i=1}^{T} P_i + \sum_{i=1}^{s} \sum_{j=1}^{t_i} p_{ij}$

□

[5] http://fit.hanu.vn/course/index.php

Table 2
Module generativity values of two CourseMan's modules in each of the three MPs.

MPs	Modules	Configuration (Step 1)				Code (Step 2)				α	M
		C	$C_1, \ldots, C_n$	$\mathcal{W}_1$	m_1	D	$D_1, \ldots, D_n$	$\mathcal{W}_2$	m_2		
MP_2	ModuleStudent	1	–	51	0.98	0	–	7500	1	0.01	0.99
	ModuleEnrolmentMgmt	0	1, 1, 1	49	0.94	0	–	7500	1	0.01	0.99
MP_3	ModuleStudent	11	–	61	0.82	72	–	7572	0.99	0.01	0.99
	ModuleEnrolmentMgmt	5	1	54	0.89	0	–	7500	1	0.01	0.99
MP_4	ModuleStudent	18	–	72	0.75	291	–	7791	0.96	0.01	0.96
	ModuleEnrolmentMgmt	17	–	61	0.72	144	–	7644	0.98	0.01	0.98

We derive from Formula 1 the following formula for m_2 (n is the number of customised descendant modules):

$$m_2 = 1 - \frac{D + \sum_{k=1}^{n} D_k}{\mathcal{W}_2} \tag{4}$$

MP_2 It can be observed in Table 1 that, for this MP, m_1 is measured based only on counting the number of customised controller configuration elements. If there exists a non-empty sub-set of these elements that require customising module actions, then m_2 is measured based on counting the LOCs of the customised components of these actions. *MP_3* It can be observed in Table 1 that, for this MP, m_1 is measured based only on counting the number of customised view configuration elements. If there exists a non-empty sub-set of these elements that require creating new view components, then m_2 is measured based on counting the LOCs of these components. *MP_4* It can be observed in Table 1 that, for this MP, m_1 is measured based on counting the numbers of customised view and controller configuration elements. If there exists a non-empty sub-set of these elements that require creating new components (view or module action), then m_2 is measured based on counting the LOCs of these components.

An important insight that we draw from our evaluation framework is that module generativity is high (which is desirable) if m_2 is high and dominates m_1. This occurs if (*i*) existing MOSA's capability (which includes the set of design patterns that have been constructed over time) is sufficient for constructing software modules in the domain and (*ii*) α is small. As explained in Section 8.1.1, condition (*ii*) typically holds in practice. Although condition (*i*) can not be guaranteed to hold for a software module whose design requirements do not match exactly with the existing MOSA's capability, it can be made to hold for other software modules that have similar design requirements. This is helped by a fact that the design patterns are highly reusable, because they are defined at the unit level (concerning *primitive* configuration elements).

Example 6. Table 2 shows the module generativity values of two CourseMan's modules (`ModuleStudent` and `ModuleEnrolmentMgmt`) that were configured for each of the three MPs discussed in this section. The last column ("*M*") lists the module generativity value.

The two master columns ("Configuration" and "Code") show statistics for step 1 and step 2 of the generativity procedure. The sub-columns of these master columns show values of each component of Formula 3 (step 1) and of Formular 4 (step 2), respectively. The two columns "$C_1, \ldots, C_n$" and "$D_1, \ldots, D_n$" list sequences of numbers for the configuration elements and LOCs (*resp.*) of the descendant modules of each module. In the current example, column "$D_1, \ldots, D_n$" does not contain any values, because none of the customisation that was made necessarily leads to code customisation in any descendant module. For instance, we initially had $D_1 = 72$ for `ModuleEnrolmentMgmt` of *MP_3*, but then discarded because the customisation could reuse the view component that was implemented as part of `ModuleStudent`. The numbers of LOCs listed in column $\mathcal{W}_1$ are counted directly from each generated MCC. The numbers in the column $\mathcal{W}_2$, however, are calculated by adding the LOCS of the customised code to the base LOCs of 7500. This is the total LOCs of the `View` and `Controller` classes that are implemented in JDomainApp.

As can be seen from the table, *M* is very high for both modules in all three MPs. This is because in all of these cases m_2 is very high and dominates m_1 (α is very small). Fig. 11 shows the CourseMan software of *MP_4*, which contains the customised views and configurations of three modules. We will use this view to illustrate both types of customisations. We will make explicit for each customisation the MPs under which it is allowed.

First, regarding to *view configuration customisation* (*MP_3*, *MP_4*), the figure shows that all three views have been configured with custom view field labels and with more complex layout. In particular, `View⟨Student⟩` is similar to the one presented in Fig. 8 in that it excludes the view field `helpRequested`. `View⟨EnrolmentMgmt⟩` has an overall tab layout, that has been customised to show the decision node structure. In this layout, all the subviews reflecting the associative fields (e.g. the subview for `EnrolmentMgmt.students`) are rendered within the tabs of a tab group. Further, the label and content of every view field has been configured to contain user-friendly strings. For instance, the view field `View⟨Student⟩.modules` is now labelled `}}Enrols Into″`. Its content now lists values (`}}IPG″`, `}}PPL″`, `}}SEG″`) of the 'bounded' domain field (`CourseModule.name`), as opposed to the `CourseModule.ids` as shown in Fig. 10.

As for *controller configuration customisation* (*MP_2*, *MP_4*), we customise the `open` action of each module with a suitable policy. We also customise this action for two descendant modules of `ModuleEnrolmentMgmt` of *MP_2*. These explain the three one values (1, 1, 1) written in the column labelled $C_1, \ldots, C_n$ in Table 2 of the module. A useful policy that we use is to automatically load all the objects from the object store. This policy is applicable when there are only a small number of objects of the concerned domain class or that these objects are required in order for the software to function. We showed in the MCC of `ModuleStudent` in Listing 1 an example of how to configure the open policy of the `open` action.

Regarding to *controller code customisation* (*MP_2*, *MP_4*), we customise the module actions in order to realise a pattern named *many-many association normaliser* [16]. This pattern involves specifying the many-many association type directly in the domain class, despite the fact that the underlying object store of this class only supports one-many association. Given that the CourseMan software uses a relational database to store objects. The domain model (shown in Fig. 1) directly supports a many-many association between `Student` and `CourseModule`. This requires the module action which handles the many-many association to conform to the pattern behaviour. This behaviour basically involves transforming the many-many association (exposed on the module's view) between the two participating domain classes into two one-many associations that are actually defined in the two classes.

More specifically, Listing 3 shows a partial MCC of `ModuleStudent` of the current CourseMan example that has been configured with two customised module actions. The customised view of this module was presented as part of Fig. 11. The two module actions are specified in the two `PropertyDesc` configurations that form the value of the property `ModuleDesc.controllerDesc.props`. They specify the customised `Create` and `Update` actions of the module. These actions are realised by the following two classes (*resp.*): `CreateObjectAndManyAssocsCommand` and `UpdateObjectAndManyAssocsCommand`.

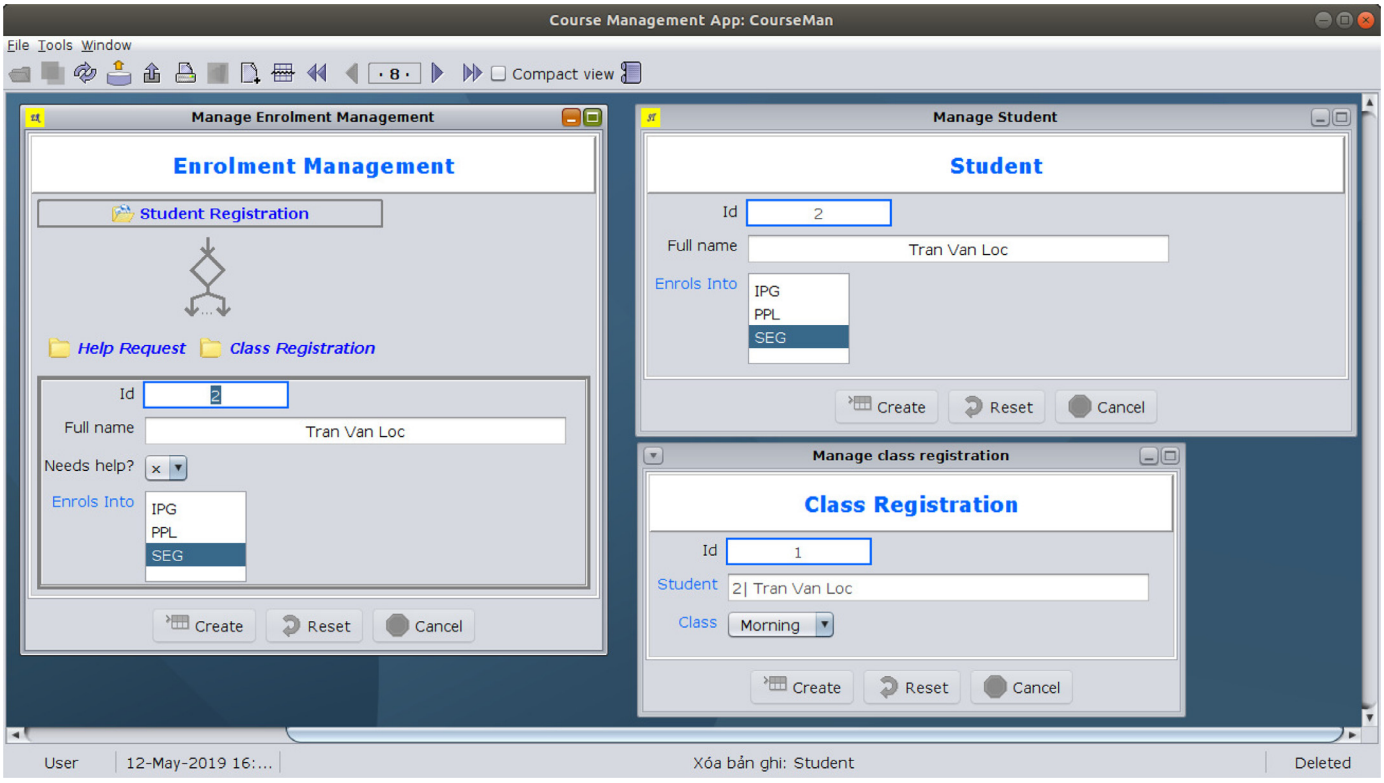

Fig. 11. An example CourseMan software that has substantially been customised.

These classes in turn implement the transformation logic described above concerning the many-many association. In Fig. 11, the actions allow the user to directly choose for each `Student` several `CourseModules` on the list.

8.2. Case study

In this section, we present a relatively complex case study, named ProcessMan (process management). The aim is to investigate how our proposed software development method (abbreviated in this section as SDM) is applied to develop software for a real-world problem domain. A key objective is to construct a process model and an MCC model that are sufficiently expressive for the domain requirements.

8.2.1. Method

According to Runeson and Höst [29], case study research (CSR) is "an empirical method aimed at investigating contemporary phenomena in their context". CSR is applicable to software engineering because software development activities are performed by human stake-holders under different conditions, especially where the boundaries between the phenomena and contexts are unclear. The type of case study that we conduct is *explanatory*. In particular, we seek a confirmation for the applicability of the SDM to relatively complex, real world environments. The subsequent subsections will report the method's steps and findings.

8.2.2. Case and subject selection

We chose the Faculty of IT (FIT) at Hanoi University[6] as the *case* for investigation. FIT is a relatively young faculty that has been undergoing significant organisational changes. One of these changes involves standardising the organisational processes in accordance with the ISO-9001:2008's quality management standard[7]. We thus chose organisational process management as the particular *subject* of our investigation.

As an educational institution, the core processes that FIT performs are teaching **subjects** (a.k.a course modules) to students every semester and formally assessing the students' performances. Conceptually, a **process** is a sequence of **task**s, each of which is a sequence of **action**s. A process is created once, by an **organisation unit**, and is periodically applied to the same unit and possibly to other organisational units that have similar needs. For certain processes, task and action would need to be specialised in order to specify more details. For example, in the assessment process, a subject is created only once but is taught in the same semester every year. This periodic delivery of a subject is called **subject-by-semester**. The type of tasks that is applied specifically to **subject-by-semester** is called **task-for-subject**. Each task-for-subject consists of a sequence of **action-for-subject**, which is a specialisation of action.

8.2.3. Data collection and analysis

Because of the nature of the SDM, data collection and analysis for the case study naturally coincide with software requirement capturing and analysis (*resp.*). We collected qualitative data, which include technical documents that describe and expert opinions about the processes. For these two data types, we applied two requirement capturing techniques: document review and semi-structured interview (*resp.*). We obtained the process documents directly from their authors, which are also members of the teaching staff. To obtain further information about the documents, we conducted informal interviews with the document authors and, in some cases, with the faculty's dean.

We analysed the requirements using our SDM, with the help of the software tool described in Section 7. In the CSR's terminology, our analysis follows the template approach [29], whereby the collected data are coded using pre-defined domain- and module-specific terms. The coded data is used to construct the domain and MCC models. Further, the analysis is performed iteratively until both models are satisfactory (as explained in Section 3).

[6] http://fit.hanu.vn/course/index.php
[7] https://www.iso.org/standard/46486.html

```
@ModuleDesc(
name="ModuleStudent",
modelDesc=@ModelDesc(model=Student.class),
viewDesc=@ViewDesc(
  //...omitted...
),
controllerDesc=@ControllerDesc(
  controller=Controller.class,
  isDataFieldStateListener=true,
  props={
    // many-many assoc commands
    @PropertyDesc(name=PropertyName.controller_dataController_create,
      valueIsClass=CreateObjectAndManyAssocsCommand.class,
      valueAsString=MetaConstants.NullValue,valueType=Class.class),
    @PropertyDesc(name=PropertyName.controller_dataController_update,
      valueIsClass=UpdateObjectAndManyAssocsCommand.class,
      valueAsString=MetaConstants.NullValue,valueType=Class.class)
  })
// ... omitted...
)
public class ModuleStudent {
  @AttributeDesc(label="Student")
  private String title;

  @AttributeDesc(label="Id",alignX=AlignmentX.Center)
  private int id;
  @AttributeDesc(label="Full name",alignX=AlignmentX.Center)
  private String name;
  @AttributeDesc(label="Needs help?",alignX=AlignmentX.Center,
      isStateEventSource=true)
  private boolean helpRequested;
  @AttributeDesc(label="Enrols Into",
  type=JListField.class,
  ref=@Select(clazz=CourseModule.class,attributes={"name"}),
  isStateEventSource=true,width=100,height=5)
  private Collection<CourseModule> modules;
}
```

Listing 3. The MCC of `ModuleStudent` with customised actions to handle many-many associations.

8.2.4. Results

Our analysis results in a domain model and an MCC model, which we will describe in this section. In addition, we will discuss threats to the validity of our approach.

Process Domain Model

Fig. 12 shows the domain model that expresses the ProcessMan's requirements. The model consists of four domain modules, which are represented in the figure as four packages. These domain modules are all named `model`, each of which is placed inside the directory structure of a software module. To ease discussion, we will refer to the domain module by the software module's name. The software module's name is written in the figure in brackets, that appear below the domain module. For example, the domain module `processstructure` (shown at the top-right of the figure) is placed in the package `processstructure.model` of the software module named `processstructure`.

As shown in the figure, the domain module `processstructure` consists of three domain classes (`Process`, `Task`, `Action`) that together describe the general structure of a process, and two other domain classes (`Task4Subject`, `Action4Subject`) that describe a specialised structure for the teaching and assessment processes. `Task4Subject` specialises `Task`, while `Action4Subject` specialises `Action`. The reason that we only specialise `Task` and `Action` and not `Process` is because class `Process` suffices for use as a common abstraction for all types of processes. At the process level, there is no real distinction between the processes.

The domain module `processapplication` has four classes: two of which (`OrgUnit`, `ProcessApplication`) represent the periodic application of a `Process`, while the other two (`SubjectTask`, `SubjectAction`) serve to resolve the two many-many associations between `Task4Subject` and `Action4Subject` and `SubjectBySemester` (*resp.*). Class `OrgUnit` also plays the role as the `Process`'s creator.

The domain module `teaching` contains two classes `SubjectBySemester` and `Subject`. The former represents the periodic delivery of the latter in a semester of each academic year. Last but not least, the domain module `hr` (which stands for "human resource") contains a class named `Teacher`. This class has a many-many association to `SubjectBySemester`, which models the ownership of each subject delivery by a teacher.

MCC Model

In ProcessMan, MCCs are created and managed within each domain module's boundary. Each domain class in a domain module is used to create one MCC. The MCCs are placed in the software module's package (which is at the same level as the module's `model` package). The owner MCC of a domain subclass is modelled as a subclass of the owner MCC of the domain superclass. This helps not only promote configuration reuse but ease configuration extensibility. To illustrate, the left hand side of Fig. 13 shows five MCCs of the domain module `processstructure`. Among these, two pairs are the subject of subclass-superclass association: `ModuleTask4Subject` is a subclass of `ModuleTask` and `ModuleAction4Subject` is a subclass of `ModuleAction`.

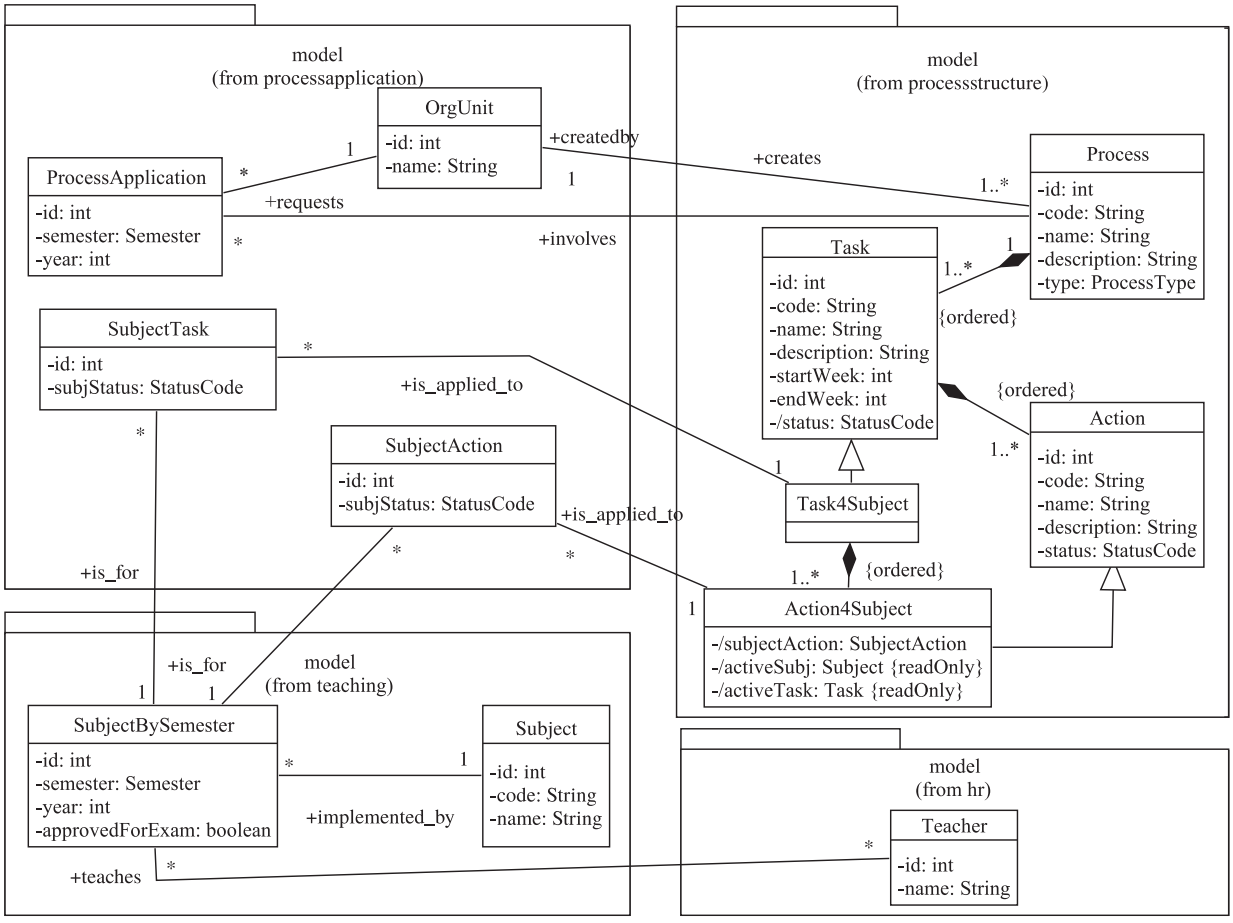

Fig. 12. PROCESSMAN's domain model.

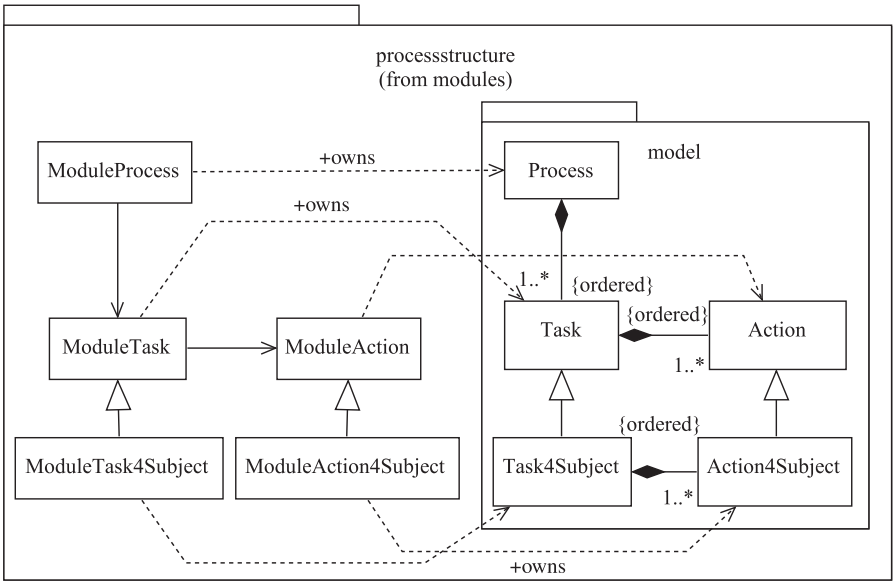

Fig. 13. A partial MCC Model for process structure.

Using the software tool of the SDM and iterative analysis, we were able to incrementally construct and verify the domain and MCC models with the domain experts. We thus conclude, with a high degree of confidence, that SDM is applicable to constructing both models for the PROCESSMAN sofware.

8.2.5. Threats to validity

We organise threats related to our case study according to the following four categories: construct validity, internal validity, external validity and reliability. ***Construct validity*** This is the extent to which "...the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions". In our case study, we have assumed that there are no misinterpretations of the process requirements that would lead to unsatisfactory representations in the models. Our SDM (see Section 3) helped mitigate the threat of misinterpretation by allowing software prototypes to be constructed quickly and incrementally from the models. Using these pro-

totypes, the domain experts can intuitively work with the technical team to verify the models against the process requirements.

Internal validity This refers to the extent to which causal relations between the investigated factors are carefully examined. The SDM, being a DDD method, assumes that a software's core is its domain model. However, there may be other aspects of the software (e.g. concerning such non-functional requirements as security, distributed processing, usability and the like) that also need to be taken into consideration. Indeed, the stake-holders involved have suggested that security and usability are two aspects that are particularly relevant to ProcessMan. Our SDM can potentially mitigate this threat because SDM (and its base architecture MOSA) is extendable to support other non-functional requirements. We will briefly discuss this capability later in Section 8.3.

External validity This is the extent to which the findings can be generalised and be of interest to other cases that have similar characteristics. There are two threats concerning this generalisation. The first threat concerns the fact that our case study was carried out over just one case. We would argue, however, that our choice of the generic process model within this case would help mitigate the threat. To the best of our knowledge, the process document was developed in accordance with the selected ISO quality management standard, making it possible to generalise the resulted models for other cases that adopt the same standard.

The second threat concerns the selected ISO standard. As of this writing, the ISO-9001:2008 has become obsolete (superseded by 9001:2015). However, the choice of ISO standard was made by the subject organisation at the time the 9001:2008 standard was still applicable. We plan to study the new standard to ascertain whether our case study's results would be affected by it.

Reliability This refers to the reliability of the data collection and analysis that were performed. Ideally, these should be reliable to the extent that if replicated (typically by another researcher) then the same result would be obtained. A threat to the reliability in our approach arises from the fact that the process document that we collected had not formally been certified by any ISO-certification agency. As such, it may not completely conform to the selected ISO standard. We would thus argue that an application of our approach to an ISO-certified process environment may require some adjustments to the data collection and/or analysis (e.g. more data would need to be collected and a combination of analysis techniques may need to be applied).

8.3. Discussion

We conclude our evaluation with two brief remarks about MOSA and MCCL. First, MCCL has been demonstrated in this paper to be usable through real-world examples. However, it would be desirable to evaluate MCCL more thoroughly, for example, by adopting the language evaluation approach of DCSL (see [11]). We plan to conduct this evaluation as part of future work. Second, MOSA and MCCL are expected to evolve in order to handle other non-functional requirements, such as security and distributed processing. In cases where a requirement type has been addressed in a separate architecture (e.g. [30,31] for security and [32,33] for distributed processing), the evolution would involve addressing the fundamental problem of MVC architectural modelling that supports other non-functional requirements. We believe that this is achievable, given a number of works that have reportedly tackled similar problems (e.g. [30,34,35]).

9. Related work

We position our work in the junctions between a number of research topics. In this section, we will review the relevant works of these topics.

9.1. DSL Engineering

There are a variety of engineering methods applicable to the development of different DSL types. A DSL can be specified based either on the domain or on the relationship with a host language (e.g. OOPL). Regarding to the domain [24], a DSL can be either vertical or horizontal. A vertical DSL, a.k.a business-oriented DSL, targets a bounded real-world domain. In contrast, a horizontal DSL (a.k.a technical DSL) targets a more technical domain, whose concepts describe the patterns that often underlie a class of vertical domains which share common features. With respect to the host language [9,36,37], a DSL can be either internal or external. In principle, internal DSL has a closer relationship with the host language than external DSL. A typical internal DSL is developed using either the syntax or the language tools of the host language. In contrast, a typical external DSL has its owns syntax and thus requires a separate compiler to process.

The term aDSL is recently coined in [10], which refers to DSLs that are internal to an OOPL and use a set of annotations to model the domain concepts. In our opinion, aDSL is an attempt to formalise the notion of fragmentary, internal DSL proposed in [9] for the use of annotation to define DSLs. We argued in [11] that using aDSL for DDD brings three important benefits for domain modelling: (1) feasibility, (2) productivity and (3) understandability.

Our proposed MCCL is a horizontal aDSL, because it can be used to create object-oriented software modules (in the DDD context) for different software domains. MCCL fits the characteristics of aDSL that is composed using annotation-based language unification [10]. However, our specification of MCCL is more precise than what is stated by Nosal et al. [10], in that it uses meta-modelling (with UML and OCL) define the language syntax. Further, MCCL specifically targets the software development domain, which Nosal et al. do not address.

9.2. Domain driven design (DDD)

Current DDD frameworks (ApacheIsIs [7] and OpenXava [8]) apply a simple form of aDSL to design the domain model. However, these work lack a method and language for software module development. Further, their MVC design, compared to ours, has a limitation in that they specify certain view-related attributes directly on the domain field. We argue that this design is a deviation from the domain model isolation principle of DDD [1].

Regarding to the choice of layered architecture for DDD, our MOSA architecture is comparable to a service-oriented architecture (SOA), named *microservices architecture* (MSA) [38]. Coincidentally, MOSA was conceived and proposed at around the same time as the emergence of MSA. It is thus worthwhile to compare and contrast these two architectures. According to Lewis and Fowler [38] MSA is "...an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API". We argue that MOSA is similar to MSA in seven out of nine fundamental characteristics of MSA that were identified by Lewis and Fowler [38]. First, both architectures are component-based. Second, the architectural components are designed to consist of all the necessary architectural layers (a feature known as broad-stack [38]). Third, both architectures adopt a domain-driven decomposition strategy, such as that employed by DDD. Both architectures support component decomposition along the boundary of domain context (formally called *bounded context* in DDD [1]). Fourth, the primary focus in both architectures is on the product, not the project [38]. Fifth, both architectures utilise, to a greatest extent possible, automation techniques [38] that help automate the development, deployment and run-time management of the components. Sixth, both architectures are designed with a failure isolation mindset. Modularity helps ensure that failures that occur in one component are compactmentised and do not severely affect the performance of other components. Seventh, both architectures enable an evolutionary design of the components. Again, modularity helps encapsulate changes that are applied to a component, allowing it to evolve with minimal impact on other components.

However, MSA differs from MOSA in three aspects, all of which stem from the fact that the two architectures have different roots. First, MSA is derived from SOA, while MOSA is from MVC. Second, although MSA

is linked to DDD, its connection is not as strong as that of MOSA. MOSA is designed specifically for DDD. Third, as a type of SOA, MSA is fundamentally decentralised. MSA components are services and communicate with each other via service interface and using a service protocol. In contrast, MOSA components are modules that communicate via an event-handling mechanism. MOSA does not (yet) support a distributed communication protocol. However, as discussed earlier in the paper, MOSA can be extended to support this NFR (and others).

9.3. Model-driven software engineering (MDSE)

MDSE is a software engineering area that overlaps with DDD in the use of models to develop software. Brambilla et al. [39] state the following two key concepts of MDSE: *models* and *transformations* (between models). These concepts form the basis for constructing DSLs as part of MDSE [24,39]. The construction involves applying the meta-modelling process to create meta-models of software modelling languages (include both general-purpose languages and DSLs). The work in [24] proposes two variants of the meta-modelling process: for vertical DSLs, it is the domain modelling process (discussed in [9]); for horizontal DSLs, it is a pattern-based modelling process. In this paper, we adopted the domain modelling process for MCCL, but targets internal, rather than external, DSL. Our method is conceived for DDD, but can also be used as part of an MDSE method: MCCL is used to construct models for the OOPL platforms.

The method in [40,41] proposes to combine DSLs in order to build a complete software model. It is a 3-step development method: (1) determine the application architecture, (2) develop the DSLs to fit this architecture and (3) define transformations between the DSLs. A key feature of this method is that the DSLs are designed to address different parts of the architecture and that each DSL is used to create not one monolithic model but multiple partial models. The adopted architecture is a layered, service-oriented architecture named SMART-Microsoft. Four DSLs are defined for this architecture: web scenario DSL (for the presentation layer), data contract DSL (data contract layer), and service and business class DSLs (business layer).

Our combined use of MCCL and DCSL in the MOSA architecture is similar in spirit to the above works. However, our approach is unique in the explicit definition of software module based on a module-based architecture for DDD. We provide a module configuration language that is used to automatically generate the modules in MOSA. Further, our DSLs are internal rather external DSLs.

9.4. OOPL support for module-based development

Shortly after publishing our recent work [13], we became aware of the Oracle's Java 9 [42] language support for code module[8]. This module sublanguage is comparable to our MCCL language. Both languages aim to increase modularity for scalable software design. The sublanguage uses the special keyword `module` to define module configuration. Each configurable element is written in the form of a code statement in the body of the module. The elements include the module's dependencies, the public packages that it wishes to make available to other modules, the services that it uses (from other modules), the services that it provides (to other modules) and the internal code structure that it wishes to make available to other modules through reflection.

However, MCCL differs from the module sublanguage in a number of ways. First, at the conceptual level, unlike the generic module concept of the sublanguage, our module class concept is specific for the problem of constructing domain-driven software. Our module consists in an MVC structure, whose model component is a domain class. Second, at the technical level, MCCL can be used to automatically generate the modules. Third, MCCL does not require a separate syntax for writing module configuration. It uses the existing Java syntax to express the configuration. This makes it easier to implement MCCL's concrete syntax in non-Java OOPLs (e.g. C# [19]), which do not support module. Another benefit is that existing language support tools can parse MCCL without requiring an update.

[8] As of this writing, the C# language does not (yet) support code module.

9.5. Attribute-oriented programming (AtOP)

Our idea of using annotation to define the concrete syntax of MCCL is inspired by attribute-oriented programming (AtOP) [43–46]. In principle, AtOP extends a conventional program with a set of attributes, which capture application- or domain-specific semantics [43], [46]. These attributes are represented in contemporary OOPLs as annotations. It is shown in [44], with empirical evidence, that because programmers' mental models do overlap the practice of recording the programmer's intention of the program elements using annotations is rather natural. Further, the use of purposeful annotations in programs help not only increase program correctness and readability [44] but raise its level of abstraction [45].

The relationship between AtOP and DDD is manifested in the use of annotations in DDD. The two DDD frameworks discussed earlier clearly demonstrate the benefits of such use. 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 [47]. This framework combines AtOP with MDSE in a top-down fashion, with an aim to define domain-specific concepts at both the modelling and programming levels. At the modelling level, the concepts are expressed in a UML-based domain specific model (DSM). At the programming level, the concepts are represented as a domain-specific code model (DSC), which are attribute-oriented programs. 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 work differs from both works [45,47] in two important ways: (1) the explicit support for software module design and (2) how MCCL, combined with DCSL, is used to automatically generate software modules.

10. Conclusion

In this paper, we tackled the problem of software module construction in domain-driven design (DDD) with a generative software module development method that uses annotation-based domain-specific language (aDSL). Our work both combined and extended a number of recent works that we have conducted on DDD with aDSL and module-based software architecture (MOSA).

The key components of our method include an aDSL, named MCCL, and a generative function (named MCCGEN) for module configurations. MCCL expresses module configuration classes (MCCs), each of which represents the configuration of a module class. The MCC's body reflects the domain class owned by the module. To ease learning and to improve productivity, we defined the MCCL's concrete syntax directly in the target OOPL using a set of annotations. We specified the MCCL's abstract syntax in UML/OCL and precisely showed how it is mapped to a conceptual model of the module configuration domain. The mapping's goal was to construct a compact and annotation-based abstract syntax model. We defined the mapping using a set of basic class model transformations. We implemented our method in a Java software framework that we have recently developed for DDD. We evaluated the applicability our method using a case study and formally analysed the MCCGEN function to show that it is correct and scalable with the domain class size. Further, we formally defined an evaluation framework for module generativity with MCCL. In this framework, we identified four generic module patterns constructiable by our method and precisely formulated the generativity extent for all the modules in each pattern. Using this framework, the development team can determine the concrete generativity values for the software modules that are constructed in their project.

We thus argue that our method makes a significant contribution to DDD, which currently lacks the support for software module and its development. Our plan for future work includes, first of all, combining our method with the current work in DCSL [11] to define a more

complete generative development methodology for DDD. This would include aDSLs to realise other architectural concerns in the MOSA model and an aDSL for configuring software from the MCCs. Related to this work would be to tackle a broader top-down modelling approach, which consists in a high-level language version of MCCL and a transformation from this to MCCL. Another future work would be to investigate how to incorporate the Java9's support for module configuration into our implementation of MCCL in the JDOMAINAPP framework.

Declaration of Competing Interest

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

Acknowledgements

This work is supported by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2015.25. We wish to thank the anonymous reviewers for their insightful comments.

Appendix A. Advanced MCC Examples and MCCGEN Analysis

A1. `ModuleEnrolmentMgmt` *with Containment Tree*

```
@ModuleDesc(
    name="ModuleEnrolmentMgmt",
    modelDesc=@ModelDesc(model=EnrolmentMgmt.class),
    viewDesc=@ViewDesc(formTitle="Manage Enrolment Management",imageIcon="
        enrolment.jpg",
        domainClassLabel="Enrolment Management",
        view=View.class,parentMenu=RegionName.Tools,topX=0.5,topY=0.0),
    controllerDesc=@ControllerDesc(controller=Controller.class),
    containmentTree=@CTree(root=EnrolmentMgmt.class,
      edges={
        // enrolmentmgmt -> student
        @CEdge(parent=EnrolmentMgmt.class,child=Student.class,
            scopeDesc=@ScopeDesc(
            stateScope={"id", "name", "helpRequested", "modules"}
          )) })
)
public class ModuleEnrolmentMgmt {
  @AttributeDesc(label="Enrolment Management")
  private String title;

  // student registration
  @AttributeDesc(label="Student Registration",
    layoutBuilderType=TwoColumnLayoutBuilder.class
    ,controllerDesc=@ControllerDesc(
      openPolicy=OpenPolicy.I
      // support many-many association with CourseModule
      ,props={
        // custom Create: to create {@link Enrolment} from the course modules
        @PropertyDesc(name=PropertyName.controller_dataController_create,
          valueIsClass=CreateObjectAndManyAssocsCommand.class,
          valueAsString=MetaConstants.NullValue,valueType=Class.class),
        // custom Update command: to update {@link Enrolment} from the course
            modules
        @PropertyDesc(name=PropertyName.controller_dataController_update,
          valueIsClass=UpdateObjectAndManyAssocsCommand.class,
          valueAsString=MetaConstants.NullValue,valueType=Class.class)
      })
  )
  private Collection<Student> students;

  // help desk
  @AttributeDesc(label="Help Request",type=DefaultPanel.class)
  private Collection<HelpRequest> helpDesks;

  // class registration
  @AttributeDesc(label="Class Registration",type=DefaultPanel.class)
  private Collection<SClassRegistration> sclassRegists;
}
```

Listing 4. The MCC of `ModuleEnrolmentMgmt` with containment tree.

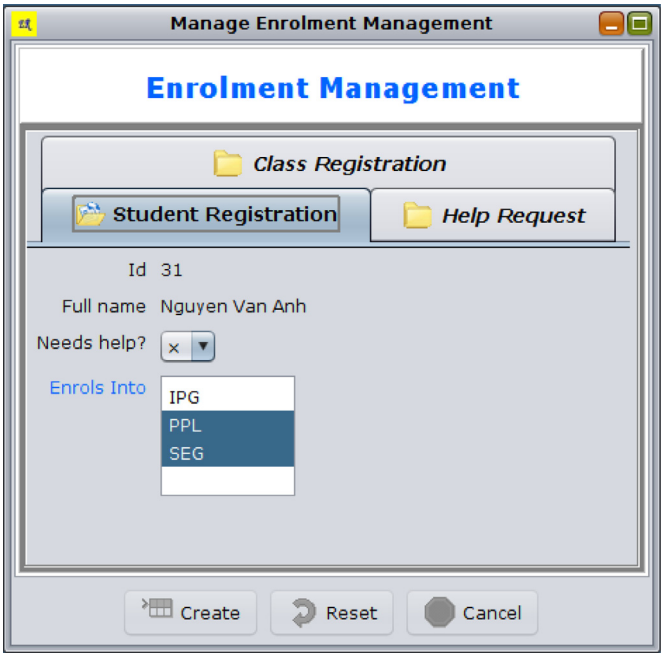

Fig. A1. The customised view of `ModuleEnrolmentMgmt` (as configured in Listing 4).

A2. `ModuleEnrolmentMgmt` *with Customised Descendant Module Configuration*

Listing 5 shows the MCC of `ModuleEnrolmentMgmt` that contains a customised containment tree. The module's view is shown in Fig. A.14. We will focus here on explaining the customisation of descendant module configuration.

Specifically, the containment tree specifies the customisation of a descendant module object of type `ModuleStudent`. The customised configuration is specified in the `CEdge` listed at lines 7–15. The containment scope of the `ScopeDesc` (line 9) specifies all four domain fields of `Student`. The `attribDescs` property (lines 12–15) specifies a customised configuration for two data fields of `ModuleStudent` that reflect `Student.name` and `id`. This configuration states that both data fields are presented by a library view class called `JLabelField`. In addition, we add, only for illustration purpose, the configuration `editable = false` in the data field for `Student.name`. This means that this data field is to be displayed as read-only, although technically in this case this configuration is redundant due to the use of `JLabelField`.

A3. Analysis of MCCGen

We discuss below our analysis of the correctness and performance of function MCCGen. We will state the theorems concerning the properties of the function and provide proofs for them.

A3.1. Structural consistency between MCC and domain class

Let us first state more precisely an important property of the MCC's body structure. This structure must comply to what we term a structural consistency rule between MCC and the domain class. The following definition makes clear what this means for MCCs and, more generally, for the overall MCC model of a software.

Definition 6 (Structural Consistency)**.** The **owner MCC** of a domain class is structurally consistent with that class if it satisfies the following conditions:

(*i*) the MCC's body consists of only the title data field and the view fields that reflect the domain fields of the domain class
(*ii*) every reference to a domain field name in the module's containment tree is a valid field name either of the domain class or of one of the domain classes of a descendant module in the actual containment tree

A **non-owner MCC** is structurally consistent with a domain class if it satisfies condition (*ii*).

An **MCC model** is structurally consistent with a domain class if all of its MCCs are structurally consistent with that class. □

Following immediately from Definition 6, we say that the MCC's body **reflects** the structure of its domain class. Let us illustrate this using the `ModuleEnrolmentMgmt`'s view presented in Fig. 3. First, the three sub-views of `ModuleEnrolmentMgmt`'s view are constructed from three view fields of this MCC. These view fields reflect three associative fields that `EnrolmentMgmt` has with `Student`, `SClassRegistration` and `HelpRequest` (see Fig. 1). The sub-view labelled "Student Registration" (whose structure is shown in the figure) contains four view fields that reflect four domain fields of `Student` that are specified in the containment scope with `ModuleEnrolmentMgmt`. For instance, the view field labelled `Id` is a data field that reflects the domain field `Student.id`.

A3.2. Correctness

The correctness of MCCGen depends on whether or not it preserves the validity and structural consistency (stated in Definition 6) of all the MCCs in the MCC model.

Theorem 1 MCCGen**.** *MCCGen correctly generates the owner MCC of the input domain class. This MCC is structurally consistent with the domain class.* □

Proof. The proof consists of two parts. First, the output MCC (m) is the owner MCC of the input domain class (c). This follows from the fact that `ModuleDesc(`m`).modelDesc` (d_o) is initialised with `model=`c.

Second, m is structurally consistent (as per Definition 6) with c. Condition (*ii*) is easily satisfied because m has the default containment tree (not created as part of Algorithm 1). Every containment scope of this tree contains all the domain fields of a child module. Condition (*i*) is satisfied due to the fact that the title field (f_d) is created as a field of m and that the `for` loop in function `AddViewFields` create other view fields (f_d) of m that reflect the domain fields f of c. Each view field has the same visibility ({}`private`″), name (n_f) and data type (t_f) as those of the corresponding domain field. □

A3.3. Performance

Performancewise, the main concern is worst-case time complexity. In the theorem that follows, we will state the worst-case time complexity of function MCCGen. To ease discussion in the proof, we will refer to this generally as complexity.

Theorem 2 (. MCCGen*Complexity)* *MCCGen has a linear worst-case time complexity of $O(\overline{F})$, where $\overline{F}$ is the number of domain fields of the input domain class.* □

Proof. The dominating part of Algorithm 1 is the invocation of function `AddViewFields` which occurs at the end of step 3. The `for` loop of this function iterates over the set F of the domain fields of c. The loop body consists of three primitive operations. Thus, the time complexity of the algorithm is big-O of $|F|$, which is $O(\overline{F})$. □

```
@ModuleDesc(
    name="ModuleEnrolmentMgmt2",
    modelDesc=@ModelDesc(model=EnrolmentMgmt.class),
    viewDesc=@ViewDesc(formTitle="Manage Enrolment Management",imageIcon="
        enrolment.jpg",domainClassLabel="Enrolment Management",view=View.class,
        parentMenu=RegionName.Tools,topX=0.5,topY=0.0),
    controllerDesc=@ControllerDesc(controller=Controller.class),
    containmentTree=@CTree(root=EnrolmentMgmt.class,
      edges={ // enrolmentmgmt -> student
        @CEdge(parent=EnrolmentMgmt.class, child=Student.class,
        scopeDesc=@ScopeDesc(
         stateScope={"id", "name", "helpRequested", "modules"}
         // custom configuration for ModuleStudent
         ,attribDescs={
           // Student.name is not editable
           @AttributeDesc(id="id", type=JLabelField.class),
           @AttributeDesc(id="name", type=JLabelField.class, editable=false),
           }))} )
)
public class ModuleEnrolmentMgmt2 {
  @AttributeDesc(label="Enrolment Management")
  private String title;

  // student registration
  @AttributeDesc(label="Student Registration",
    layoutBuilderType=TwoColumnLayoutBuilder.class
    ,controllerDesc=@ControllerDesc(
      openPolicy=OpenPolicy.I
      // support many-many association with CourseModule
      ,props={
        // custom Create: to create {@link Enrolment} from the course modules
        @PropertyDesc(name=PropertyName.controller_dataController_create,
         valueIsClass=CreateObjectAndManyAssocsCommand.class, valueAsString=
             MetaConstants.NullValue, valueType=Class.class),
        // custom Update: to update {@link Enrolment} from the course modules
        @PropertyDesc(name=PropertyName.controller_dataController_update,
         valueIsClass=UpdateObjectAndManyAssocsCommand.class, valueAsString=
             MetaConstants.NullValue, valueType=Class.class)
      })
  )
  private Collection<Student> students;

  // help desk
  @AttributeDesc(label="Help Request",type=DefaultPanel.class)
  private Collection<HelpRequest> helpDesks;


  // class registration
  @AttributeDesc(label="Class Registration",type=DefaultPanel.class)
  private Collection<SClassRegistration> sclassRegists;
}
```

Listing 5. The MCC of `ModuleEnrolmentMgmt` with custom subview configuration for a descendant module of type `ModuleStudent`.

References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.
[2] V. Vernon, Implementing Domain-Driven Design, 1 edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2013.
[3] B. Meyer, Object-Oriented Software Construction, 2nd, ISE Inc., Santa Barbara (California), 1997.
[4] B. Liskov, J. Guttag, Abstraction and Specification in Program Development, MIT Press, 1986.
[5] B. Liskov, J. Guttag, Program Development in Java: Abstraction, Specification, and Object-Oriented Design, Pearson Education, 2000.
[6] OMG, Object Management Group, OMG Unified Modeling Language (OMG UML) version 2.5, 2015, https://www.omg.org/spec/UML/2.5/PDF.
[7] D. Haywood, Apache isis - developing domain-driven java apps, Method. Tool. Pract. Knowl. Sour. Softw. Develop. Prof. 21 (2) (2013) 40–59.
[8] J. Paniza, Learn OpenXava by Example, CreateSpace, Paramount, CA, 2011.
[9] M. Fowler, T. White, Domain-Specific Languages, Addison-Wesley Professional, 2010.
[10] M. NosÃ!'lâ, M. SulÃr, J. JuhÃ!'r, Language composition using source code annotations, Comput. Sci. Inform. Syst. 13 (3) (2016) 707–729.
[11] D.M. Le, D.-H. Dang, V.-H. Nguyen, On domain driven design using annotation-based domain specific language, Comput. Lang. Syst. Struct. 54 (2018) 199–235.
[12] 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.
[13] 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.
[14] G.E. Krasner, S.T. Pope, A description of the model-view-controller user interface paradigm in the smalltalk-80 system, J. Obj.-Orient. Program. 1 (3) (1988) 26–49.
[15] K. Czarnecki, Overview of generative software development, in: J.-P. Banâtre, P. Fradet, J.-L. Giavitto, O. Michel (Eds.), Unconventional Programming Paradigms, LNCS, Springer Berlin Heidelberg, 2005, pp. 326–341.
[16] D.M. Le, D.-H. Dang, V.-H. Nguyen, Domain-driven design patterns: ametadata-based approach, in: Proc. 12th Int. Conf. on Computing and Communication Technologies (RIVF), IEEE, 2016, pp. 247–252.
[17] D.M. Le, jDomainApp version 5.0: A Java domain-driven software development framework, Technical Report, Hanoi University, 2017.
[18] J. Gosling, B. Joy, G.L.S. Jr, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st, Addison-Wesley Professional, Upper Saddle River, NJ, 2014.
[19] A. Hejlsberg, M. Torgersen, S. Wiltamuth, P. Golde, The C# Programming Language, 4 edition, Addison Wesley, Upper Saddle River, NJ, 2010.
[20] D.M. Le, D.-H. Dang, V.-H. Nguyen, Generative software module development for domain-driven design with annotation-based domain specific language, Technical Report, VNU University of Engineering and Technology, 2019.
[21] A. Barišić, V. Amaral, M. Goulão, Usability driven DSL development with USE-ME, Comput. Lang. Syst. Struct. 51 (2018) 118–157.
[22] P. Hudak, Modular domain specific languages and tools, in: Proc. 5th Int. Conf. on Software Reuse, 1998, pp. 134–142.
[23] R.F. Paige, D.S. Kolovos, F.A.C. Polack, A tutorial on metamodelling for grammar researchers, Sci. Comput. Program. 96 (2014) 396–416.
[24] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st, Addison-Wesley Professional, Upper Saddle River, NJ, 2008.
[25] D.M. Le, MCCL implementation in jDomainApp, 2017, VNU-DSE, https://github.com/vnu-dse/mccl.
[26] F. Tomassetti, D. van Bruggen, N. Smith, JavaParser, 2016, http://javaparser.org/.
[27] D.M. Le, D.-H. Dang, H.T. Vu, jDomainApp: a module-based domain-driven software framework, Proc. 10th Int. Symp. on Information and Communication Technology (SOICT), ACM, 2019 https://doi.org/10.1145/3368926.3369657.
[28] E. Evans, Domain-Driven Design Reference: Definitions and Pattern Summaries, Dog Ear Publishing, LLC, 2014.
[29] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131.
[30] A.V. Uzunov, K. Falkner, E.B. Fernandez, Decomposing distributed software architectures for the determination and incorporation of security and other non-functional requirements, in: Proc. 22nd Australian Software Engineering Conf. (ASWEC), 2013, pp. 30–39.
[31] J.E. Hachem, Z.Y. Pang, V. Chiprianov, A. Babar, P. Aniorte, Model driven software security architecture of systems-of-systems, in: Proc. 23rd Asia-Pacific Software Engineering Conf. (APSEC), 2016, pp. 89–96.
[32] R. Amir, A. Zeid, A UML profile for service oriented architectures, in: Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications, in: OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 192–193.
[33] F. Rademacher, S. Sachweh, A. Zündorf, Towards a UML profile for domain-driven design of microservice architectures, in: Proc. Workshop Software Engineering and Formal Methods, in: LNCS, Springer, Cham, 2017, pp. 230–245.
[34] D. Distante, P. Pedone, G. Rossi, G. Canfora, Model-driven development of web applications with UWA, MVC and JavaServer Faces, in: Proc. 7th Int. Conf. Web Engineering, in: LNCS, Springer, Berlin, Heidelberg, 2007, pp. 457–472.
[35] I. Marsic, An architecture for heterogeneous groupware applications, in: Proc. 23rd Int. Conf. on Software Engineering, in: ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 475–484.
[36] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, SIGPLAN Not. 35 (6) (2000) 26–36.
[37] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[38] J. Lewis, M. Fowler, Microservices, 2014, https://martinfowler.com/articles/microservices.html
[39] M. Brambilla, J. Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice, 1st, Morgan & Claypool Publishers, 2012.
[40] J. Warmer, A model driven software factory using domain specific languages, in: Proc. Euro. Conf. Model Driven Architecture- Foundations and Applications, Springer, Berlin, Heidelberg, 2007, pp. 194–203.
[41] J. Warmer, A. Kleppe, Building a Flexible Software Factory Using Partial Domain Specific Models, 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), 2006.
[42] J. Gosling, B. Joy, G.L.S. Jr, G. Bracha, A. Buckley, D. Smith, The Java Language Specification: Java SE 9 Edition, Oracle America, Inc., California, USA, 2017.
[43] V. Cepa, S. Kloppenburg, Representing Explicit Attributes in UML, in: Proc. 7th Int. Workshop on Aspect-Oriented Modeling (AOM), 2005.
[44] M. Sulír, M. Nosál, J. Porubän, Recording concerns in source code using annotations, Comput. Lang. Syst. Struct. 46 (2016) 44–65.
[45] M. Balz, Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-Based Software Development, Universität Duisburg-Essen, 2012 Ph.D. thesis.
[46] V. Cepa, Attribute Enabled Software Development, VDM Verlag, Saarbrücken, Germany, Germany, 2007.
[47] H. Wada, J. Suzuki, Modeling Turnpike frontend system: a model-driven development framework leveraging UML metamodeling and attribute-oriented programming, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, LNCS, Springer Berlin Heidelberg, 2005, pp. 584–600.