UDML: On Transforming a Unified Domain Model to an Executable for Domain-Driven Design

Van-Vinh Le
Department of Software Engineering
University of Engineering and Technology
Vietnam National University, Hanoi
Vinh University of Technology Education
levanvinh@vuted.edu.vn

Duc-Hanh Dang
Department of Software Engineering
University of Engineering and Technology
Vietnam National University, Hanoi
hanhdd@vnu.edu.vn

***Abstract*—Domain-driven design (DDD) aims to iteratively develop software around a realistic domain model. Recent research in DDD has been focusing on a methodology with internal DSLs represent the domain model using annotation-based domain-specific language (aDSL), specifically using the activity graphic language (AGL) [1], the domain class specific language (DCSL) [2], the module configuration class language (MCCL) [3]. They have some limitations; for instance, aDSL can make the code difficult to read and understand for communication between stakeholders.**

This paper aims to develop a domain-specific language (DSL) called UDML, which precisely represents a unified domain modeling language that unifies DSLs such as DCSL, AGL, and MCCL. It provides a transformation-based method to automatically generate software artifacts with a new type of presentation for the domain model. The new form is an external DSL using metamodeling to perform and identify the relationship between external DSL and internal DSL, ensuring consistency and overcoming the limitations of internal DSL. We define a metamodel to capture the technical domain of AGL, DCSL, and MCCL for UDML's abstract syntax. Additionally, we provide a textual concrete syntax for this language. Furthermore, we address the task of ensuring consistency between the unified models (AGL, DCSL, and MCCL) and the Java unified model by defining a model transformation using rules with Acceleo.

This work introduces a complete software development technique that leverages an external DSL with defined transformation rules. We also implement tool support for this language and perform an evaluation.

***Index Terms*—AGL, DCSL, DDD, DSL, MCCL, Metamodeling, Internal DSL, External DSL**

I. Introduction

Domain-driven design (DDD) [4] aims to iteratively develop software around a realistic model of the problem domain. DDD often needs to have many different forms of representation: (Communicate effectively to different stakeholders, capture different aspects of the domain, support different software development activities such as representation in different levels of abstraction and increase communication between stakeholders, etc.). We support the methodology DDD and we focus on DDD to research to solve DDD challenges such as precisely modeling the business domain because business domains are often complex and ever-changing, with many detailed rules, processes, and relationships, communication between stakeholders. For example, consider the two most common representations: high-level (UML/OCL) and programming language level (internal DSL). In both cases, the domain model is specified using Java at the technical/programming level.

In DDD, a form of annotation-based language extension of OOPL has been used to develop software frameworks like ApacheIsis [5], [6] and OpenXava [7]. The design rules in a domain model are expressed by a set of annotations. However, existing DDD works have several limitations: they do not formalize their extensions into a language; their extensions, although including many annotations, do not identify the minimal set of annotations that express the domain model; and they do not investigate what DSLs are needed to build a complete software model and how to engineer such DSLs. In his book [4], Evans does not explicitly consider behavioural modeling as part of DDD. This shortcoming remains, inspite of the fact that the method considers object behaviour as an essential part of the domain model and that UML interaction diagrams would be used to model this behaviour. In UML [8], interaction diagrams (such as sequence diagram) are only one of three main diagram types that are used to model the system behaviour. The other two types are state machine and activity diagram. More generally in practice, UML activity diagram has been argued to be suitable for domain experts to use [9]. The high-level specification UML activity and class diagram is one of the behavioral models used for modeling the global behavior of systems, meta modeling for DSL [10] to construct conceptual model of the domain as a UML/OCL class diagram using abstract syntax model suitable for embedding in to a host object oriented programming language (OOPL). The UML [8] involves the construction of an object-oriented model of the abstract syntax optionally, the concrete notation and semantics of the target language [11]. To the best of our knowledge, although the existing works in DDD [4], [5], [12], [13] support domain module, they do not address how to use it to form software module. Further, they do not consider modules as first-class objects and, thus, lack a method for their development. Not only that, they do not precisely characterise the software that is developed from the domain model. These shortcomings would surely hinder DDD adopters in their efforts to apply the method in practice. In the [14] MOSA realises the DDD's layered architecture. It positions the domain model at the core and (currently) incorporates the UI concern into a layer around this core. A software module in MOSA forms a micro, functional software. In our recent work [3], with this language, we construct software modules directly from the domain model and automatically generate these modules. We also leverage domain-specific modeling (DSM), which is a software engineering approach, to design and develop software systems called the JDA method. The JDA method proposes a number of internal DSLs, including the following: AGL [1], DCSL [2], MCCL [3], to represent the domain model using annotation-based domain-specific language (aDSL). It also provides some limitations. For instance, aDSL can make the code difficult to read and understand for communication between stakeholders. When changing the request, modifying the use of aDSL can be complicated and time-consuming. Additionally, aDSL often lack flexibility compared to handwritten code. An external DSL is a specialized programming language designed to solve problems in a specific domain. It offers several advantages, including transparency, reusability, maintainability, and ease of integration, which ultimately contribute to improved communication within development teams, enhanced software development efficiency, and various other benefits [15], [16]. This motivates the need for a new representation of external DSLs, as they address the limitations of internal DSLs in terms of readability, maintainability, error reduction, scalability, and integration with other systems and tools, etc. The paper [17] proposes techniques to ensure consistency between models in the multi-model independent software development approach to represent all aspects at different stages of the development process and to support incremental consistency analysis techniques for later evolution of the new model version [18], [19]. However, there is still a lack of identification of the relationship between external DSLs and internal DSLs to ensure consistency, which would provide developers, analysts, and users with a unified view of the system, facilitating communication, collaboration, and system maintenance.

In this paper, we propose a new type of presentation for the domain model. The new form is an external DSL using metamodeling. We focus on defining three metaconcepts: (1) Domain class specific language (DCSL), (2) Incorporated domain class and behavior (AGL), and (3) Software architectural design level (MCCL). We identify the relationship between external DSL and internal DSL to ensure consistency. This helps developers, analysts, and users have a unified view of the system, facilitating communication, collaboration, and system maintenance. Additionally, we implement tool support for this language and perform an evaluation.

Our approach involves developing a DSL called UDML (Unified domain modeling language) and providing a transformation-based method to automatically generate software artifacts from a metamodel using Acceleo. We define a metamodel to capture the technical domain for UDML's abstract syntax and then provide a textual concrete syntax for this language. Additionally, we define the rules for transforming unified domain model to Java. Finally, we implement tool support for this language and perform an evaluation of its performance.

To summarize, the main contributions of this paper are as follows:

- Add a new domain model specification, using external DSL
- Ensure consistency between external DSL and internal DSL using model transformation techniques.
- Perfecting and increasing automation for the JDA method
- Tools to support and experiment with the proposed method

The rest of the paper is organized as follows: Section II surveys related work. Section III explains the basic idea of our approach. Section V specifying a unified domain model providing a formal syntax and semantics for it. Section V defining the transformation unified domain model to Java (UDM2Java). A tool support and experiments are explained in Section VI evaluates our language. This paper is closed with conclusions and a discussion of future work Section VII concludes the paper.

II. Related Work and Background

We position our work at the intersection between the following areas: DSL engineering, DDD, Domain-specific modeling (DSM), model-driven software engineering (MDSE), activity graph language (AGL), domain class specific language (DCSL) and module configuration class language (MCCL).

DSL Engineering. Domain-specific languages (DSLs) can be defined based on either the domain itself or the relationship with a host language (e.g. OOPL). In terms of the host language [10], [15], [20]–[23], DSLs can be categorized as either internal or external. Internal DSLs have a closer connection to the host language and are typically developed using the host language's syntax or language tools. In contrast, external DSLs have their own distinct syntax and require a separate compiler for processing. Recently, the term aDSL has emerged, referring to DSLs that are internal to an OOPL and utilize annotations to model domain concepts. This concept formalizes the idea of fragmentary, internal DSLs proposed in previous work [3].

In the recent works [2], [3], using the DSLs: DCSL, AGL, MCCL for DDD has demonstrated several benefits for domain modeling: (1) Feasibility, (2) Productivity, (3) Understandability. The aDSL 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 [20] for the use of annotation to define DSLs. However, aDSL also has limitations. It can make code less readable and hinder communication among stakeholders. Additionally, aDSLs may lack flexibility compared to handwritten code. Accurately representing the domain model remains crucial for successful DDD, leading to enhanced understanding, improved system features, easier maintenance, and increased reusability. When exploring external DSLs [15], [16], we find that they offer a convenient way to specify conversion / ensure consistency between different forms of specifications: (1) High-level specifications (natural language, UML diagrams, etc.), (2) Java-level specifications, (3) GUI-based specifications. External DSLs can effectively specify metamodels for AGL, DCSL, and MCCL.

DDD. Domain-driven design (DDD) [4] aims to develop complex software around a realistic domain model. This model is constructed with a focus on domain knowledge, aiming for unification to enable effective communication throughout the entire software development process. The core idea in DDD is to model software that aligns with the specific problem domain, by leveraging insights from domain experts [13]. Several studies [20], [24] have advocated for combining DDD with DSL to elevate the level of abstraction in the target code model. Current DDD frameworks (ApacheIsIs [5], [6] and OpenXava [7]) apply a simple form of aDSL to design the domain model and support domain modules. However, these frameworks lack methods to utilize aDSL for forming software modules and their development.

In this paper, we extend the DDD method [4] to construct a unified domain model. We combine this with an activity graph model to operate in a module-based software architecture and implement it in our external DSL.

MDSE. Model-driven engineering (MDE) [25] emphasizes the proactive utilization of model transformations to facilitate various stages of the software development process. Consequently, models and their transformations emerge as pivotal artifacts throughout the software development lifecycle. In practice, real-world software development often becomes convoluted, affecting project management, incurring costs, and impacting development time [26]. Meanwhile, development efforts strive to reduce time and costs while meeting the diverse needs of software across various domains of human life. The concept of combining Model-driven software engineering (MDSE) [27] with DSLs [22] involves applying the metamodeling process to create metamodels for software modeling languages. Our method shares similarities with the approach proposed by Van Gurp and Warmer, which employs a combination of DSLs to construct a comprehensive software model. We introduce a new DSL called unified domain modeling language (UDML), which unifies the DSLs: DCSL, AGL, and MCCL. UDML, along with formal operational semantics, enables us to create a unified domain model consistent with a unified model (Java) using model transformation rules with Acceleo. This aims to facilitate automated transformations that generate software artifacts from UDML. DSM is automating the creation of executable source code directly from the DSL models, the language-based approach to raise the level of abstraction in order to speed up development work and set variation space already at specification and design phase [21]

Unified modeling with UML diagrams. Recent research has attempted to combine UML structural and behavioral diagrams to construct a system model, similar in spirit to the unified domain model that we proposed in this paper. The work in [28], [29] discusse combining UML class and state machine diagrams to model the system, the work in [30] discusses combining UML Class and Sequence diagrams. Our proposed unified domain modeling is novel in that it combines UML class and activity diagrams by incorporating the domain-specific structure (activity classses) into the class diagram for a unified class model. The unified class model and activity graph are connected by virtue of the fact that nodes in the graph execute actions of the modules that own the domain classes in the model.

In our recent work, we incorporated domain behaviors into a domain model that encompasses both structural and behavioral aspects of the domain [1]. The approach involved building a unified class model with an annotation internal DSL to attach features of the host programming language (such as Java) within a domain-driven architecture. This was done to bridge the gap between the domain model and its implementation. A pattern-based approach was employed, where domain behaviors are specified using UML activity diagrams with basic constructs that correspond to the five essential activity modeling patterns. However, when attempting to construct a concrete syntax for the entire problem and combine it with the complete module architecture, several challenges arise. Manual construction of such a model can be quite tedious.

In this paper, our approach involves defining an external DSL for AGL with modeling aspect behavior for development and integrating patterns represented by an internal DSL (Java, textual) to facilitate easier software construction through model transformations, allowing for easy adjustment to suit specific needs. The high-level specification UML activity and class diagram is one of the behavioral models used for modeling the global behavior of systems, meta modeling for DSL [10] to construct conceptual model of the domain as a UML/OCL class diagram using abstract syntax model suitable for embedding in to a host object oriented programming language (OOPL). The UML [8] involves the construction of an object-oriented model of the abstract syntax optionally, the concrete notation and semantics of the target language [11].

While there have been numerous studies on various representation and specification techniques, there remains a lack of domain-based representation techniques that address the following aspects: (1) covering both structural and behavioral aspects, (2) providing comprehensive information specification to ensure natural software generation, and (3) being adaptable while maintaining quality and adhering to design requirements.

III. Overview of Our Approach

This section explains our basic idea a visual process from the design stage to quickly create software. The system will automatically execute the domain requirements, using approaches for generating software aligned with DDD.

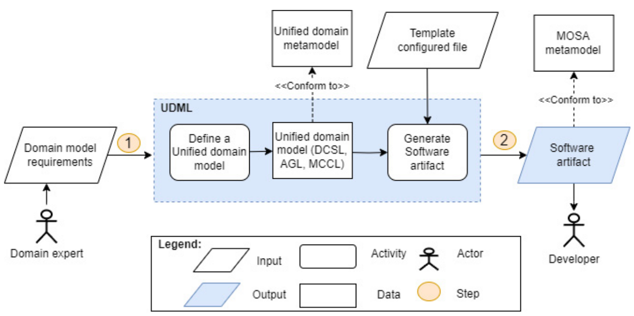

Fig. 1. Overview of our approach

Figure 1 shows the main steps of the UDML approach. This technique conceptually involves performing two steps. Our goal is to address the challenges as follows.

Unified domain model: To define precise semantics for the unified domain model by characterizing its execution using a new domain model specification through an external DSL.

Generate software artifact: To ensure consistency between external and internal DSLs, model transformation techniques can be employed to automatically generate software artifacts.

In our approach, step 1 takes input in the form of a specification domain requirements capturing the system's domain concepts. Additionally, a template configuration file about aDSL (AGL, DCSL, and MCCL) extends the conventional DDD's domain model with the executable domain model [4].

Step 2 defines a unified domain model and provides the foundation for transformations that automatically generate software artifacts from UDML. UDML operates in two substeps:

- *Define a unified domain model:* To create a unified domain model (DCSL, AGL, MCCL), UDML analyzes the input domain model and unifies it into a unified domain model conforming to its metamodel (presented in Section IV).
- *Generate software artifacts:* For each unified domain model and the configured file, UDML utilizes Acceleo with transformation rules for model transformation, generating a software artifact (presented in Section V). This software, implemented in DDD as a GUI and module-based application using the JDA framework.

IV. Specifying a Unified Domain Model

In this section introduces a DSL called UDML to unify DSLs such as DCSL, AGL, and MCCL, resulting in a precise representation of a unified domain modeling language. The objective is to improve productivity and allow designers to easily construct correct artifact software.

A. Defining a unified domain model

In this session, we define the metaconcept: AGL, DCSL and MCCL.

1) The abstraction syntax of AGL: The metaconcept of AGL includes an external DSL with modeling aspect behavior for development (the purpose is to represent specific syntax graphically) and integrating patterns represented by an internal DSL (Java, textual) to facilitate easier software construction through model transformations.

Figure 2 is metamodel of AGL, the entire AGL metamodel can be defined using a $RootNode$ that references both the $ActivityGraph$ and $ModuleAct$ with an $actSeq$ association. The $ActivityGraph$ represents an activity model that incorporates domain behaviors into a domain model. It includes $Nodes$ and $Edges$, along with an action $Module$. The $Nodes$ are referenced to the domain class through the $refCls$ link. These $Nodes$ are control nodes corresponding to the Controll class, which is represented as patterns. Finally, an activity-specific association links the activity class and the control Nodes.

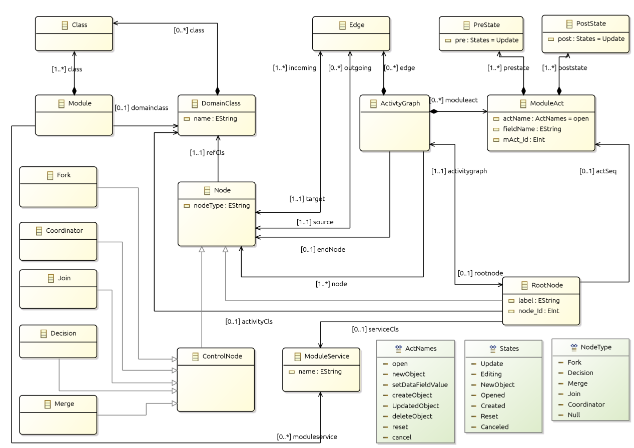

Fig. 2. metamodel of AGL

2) The abstraction syntax of DCSL: Figure 3 is metamodel of DCSL, which is represented through metaconcepts. A domain class is composed of two parts: a metaconcept class and a constraint captured by an annotation named $DClass$. Similarly, a domain field is composed of a meta-concept field along with a set of state space constraints. These constraints are represented by an annotation named $DAttr$. An Associative field is a specific type of domain field. It represents one end of an association between two domain classes. Finally, a domain method combines a method with a certain behavior. This behavior is captured by an annotation named *DOpt* and optionally another annotation named $AttrRef$. In the UDML specification, domain classes are used to define configured unified domain model templates.

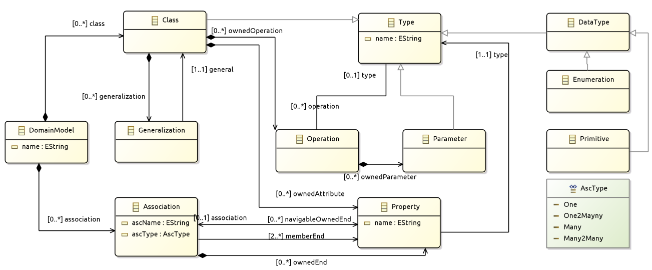

Fig. 3. metamodel of DCSL

3) The abstraction syntax of MCCL: Figure 4 is metamodel of MCCL, which is specified through several classes: $ModuleConfig$: Represents the configuration of a module. $ModelConfig$, $ViewConfig$, and $ControllerConfig$: Represent the configurations of the three main components within a module. The module containment tree is represented by the association from $ModuleConfig$ to classes tree: $Node$, $RootNode$, and $Edge$. Class node has an attribute named $DClass$, specifying the domain class of the corresponding module. $RootNode$ inherits from Node and includes an additional attribute named stateScope. Each connection in the UDML metamodel consists of two classes at its ends, playing different roles. For each property describing a role on one class, there will always be a corresponding property on the opposite class, represented by the opposite relationship.

B. UDML's abstract syntax

The UDML created using the syntax of AGL, DCSL, and MCCL maps to the syntax of MOSA [3]. Figure 5 shows the metamodel of a unified DCSL, AGL, and MCCL metamodels, which is represented through metaconcepts $Software$ and $Configure$. This is specified through several classes: (1) $Software$: This class has relationships to the *ActivityGraph* of the AGL metamodel and the $Module$ of the MCCL metamodel. Specifying the domain class of the corresponding module is done through the *Class* of the DCSL metamodel, which has a relationship to

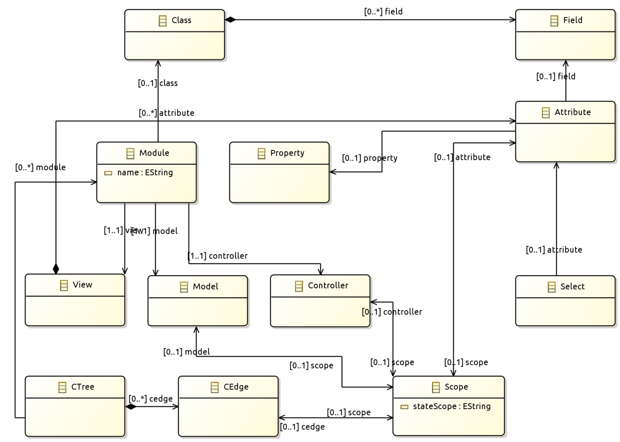

Fig. 4. metamodel of MCCL

the $Model$ of the MCCL metamodel, and (2) $Configure$: This class relates all software artifacts by configuration files.

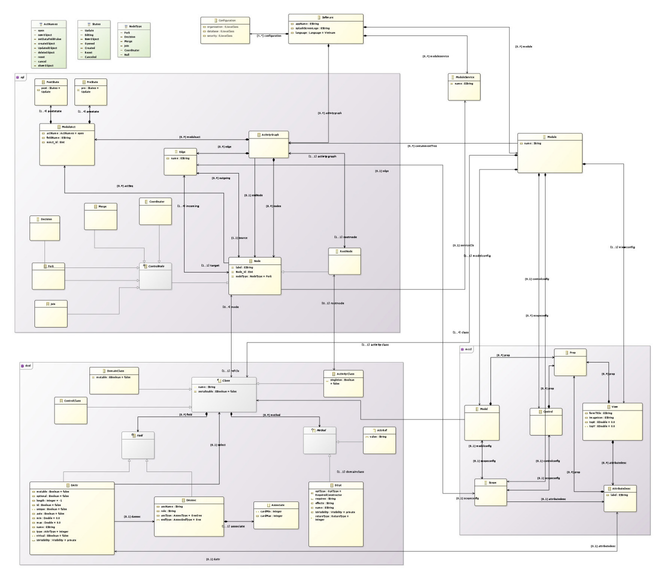

Fig. 5. metamodel of UDML

An object model, typically depicted as a class diagram, is often supplemented by OCL conditions that define properties or constraints on the domain. For UDM models, we also use OCL to help improve the domain model verification and testing process by allowing the definition of conditions that must be satisfied. OCL is a formal language characterized by the following features: its expressions, whether they are object constraints or queries, do not produce side effects. Additionally, OCL is a typed language, meaning every valid (well-formed) OCL expression has a type corresponding to the type of its evaluated value. For example, we illustrate below how a number of common invariant constraints on the well-formedness rules for AGL concerning *ActivityGraph* is as follows:

`ActivityGraph`

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

C. UCML's concrete syntax

The specific syntax of UDML has not been precisely defined. We employ textual internal DSL (aDSL) syntax and incorporate corresponding conversion rules to ensure internal DSL compliance with the designed metamodel. The scope of the paper focuses on the abstract syntax. However, we define the external DSL for AGL to incorporate behavioral aspects into domain models, with specific syntax represented graphically as shown in Figure 10:

We perform mapping between input domain requirements and a unified domain model according to the following requirements:

1) Each activity node instance in the source unified domain model is mapped to an instance of the same name in the target AGL model.
2) Each node connection (edge) in the unified domain model is labeled and mapped to the *OutClass* instance of the AGL model in the target model.
3) Each control node instance in the source unified domain model is mapped to a corresponding Pattern model (Decision, Fork, Merge, Join, Sequential, Coordinate) with the AGL model instance of the same name.
4) Each behavior refers to the domain class by the node through the structure of the domain model. This ensures that the execution state of the activity is synchronized with the current state of the unified domain model.

Mapping rule $Node$ is an identity map on `Decision`, `Join`, `ActName`, and `State`. These metaconcepts are transferred directly to concrete syntax in graphical form. Given the additional OCL constraints that were defined previously on `MAct`, mapping rule $MAct$ maps `ModuleAct` to `MAct`. Mapping rules *ANode* and *ANode.init* define the mapping for `ANode`. Specifically, $ANode$ maps `Node` to the field set of `ANode` that excludes (`excl.`) three fields (`nodeType`, `outClses`, and `init`).

The inside *ActivityGraph* is a set of (A, S, D, C, As) in which: A represents activity class that realizes the domain class representing the activity. S is $(a_1,...,a_n)$, a module action if a_i.postStates $\subseteq$ a_{i+1}.preStates ($\forall$ a_i,a_{i1} $\in$ S). D represents the data class that realizes the domain class representing each data store. C represents the control class that captures the domain-specific state of a control node. A control class representing a control node is named after the node type: decision class, join class, merge class, and fork class. *As* represents activity-specific association that realizes the association between each class pair: A and C (a merge class), A and C (a fork class), A and D (which does not represent the data store of an action node connected to either a merge or fork node), C (a merge or fork class), and D (which represents the data store of an action node connected to the merge or fork node).

V. Defining the Transformation UDM2Java

In this section, we address the task of ensuring consistency between the unified models (AGL, DCSL, and MCCL) and the Java unified model. We achieve this by defining a model transformation using rules with Acceleo.

Rule 1: Map classes, attributes, and class methods in the model's domain class to the corresponding *Class* and *Attribute* in Java source code.

Rule 2: Map module classes, define model configurations (or "configure model elements" or "generate model configurations"), controllers, and views, and describe properties.

Rule 3: Map nodes inside the *ActivityGraph* in UDM to Java source code, ensuring consistency in the source code.

Rule 4: Non-coordinator control nodes in the activity domain diagram will not have a module of their own, but will instead be located in the controls folder within the module of the activity class.

Rule 5: Map high-level software configuration specifications from the model to Java annotations.

Algorithm 1 shows generates a software artifact

Algorithm 1: SA- Transformation UDM to software artifact algorithm

input : UDM and T the template configured unified model
output : SA - Software artifact

1 The start the unified domain model; Depend on the elements call the T corresponding module
2 For each domain class of unified domain model do
3 $SA \leftarrow T$: classes, attributes, and class methods
4 For each model configurations do
5 $SA \leftarrow T$: module classes
6 for each node do
7 $SA \leftarrow T$: ActivityGraph
8 if Non-coordinator control nodes then
9 $SA \leftarrow T$: control pattern
10 For each software configuration do
11 $SA \leftarrow T$: software configuration
Return SA

In lines 2 and 3 of Algorithm 1, the **Rule 1** T will map classes and name attributes, as well as feature method attributes in the UDM, to $Classes$, $Attributes$, and $Methods$ corresponding to the Java source code:

- *Name of domain class DClass:* (1) Each domain class defined in the model will be converted to a Java class name. (2) Java class files are named after domain model classes to ensure consistency. (3) Names are initialized with parameters, if any.

- *Name of attribute DAttr:* (1) Attributes in a Java class include: attribute name, return type name (String, Boolean, Integer, etc.), and attribute access scope (public, protected, or private). (2) Get() and Set() methods. (3) *DAttr* annotation is added to each attribute to describe other information.
- *Name of method DOpt:* (1) Methods in a Java class include: method name, return type name (String, Boolean, Integer, etc.), return parameters, and method access scope (public, protected, or private). (2) *DOpt* annotation is added to each method to describe other information.
- *Name of association DAssoc:* (1) The *DAssoc* annotation is added to describe an attribute's relationship information with other domain classes. (2) The *DAssoc* annotation appears only when the attribute is of the form Domain or Collection using an if/else statement.

Figure 6 shows a part of the Acceleo code to map class and property names, as well as type methods and properties

dcsl_generate.mtl

```
[comment encoding = UTF-8 /]
[module dcsl_generate('http://www.example.org/dcsl', 'http://www.eclipse.org/emf/2002/Ecore')]
[template public generateElement_dcsl(class : Class)]
public class [class.name/] {
private static int idCounter = 0;
    [for (dattr : dcsl::Field | class.field->select(f | f.oclIsKindOf(dcsl::DAttr)))]
        [let dattrCasted : dcsl::DAttr = dattr.oclAsType(dcsl::DAttr)]

[if (dattrCasted.type = dcsl::AttrType::Collection) ]
    @DAttr(name = "[dattrCasted.name/]",attrType=[dattrCasted.type/],
                filter=@Select(clazz=[dattrCasted.select.name/].class))
    @DAssoc(ascname="[dattrCasted.dassoc.ascName/]",role="[dattrCasted.dassoc.role/]",
                asocType=[dattrCasted.dassoc.ascType/],endType=[dattrCasted.dassoc.endType/],
    associate=@Associate(attrType=[dattrCasted.select.name/].class,cardMin=[dattrCasted.dassoc.associate.cardMin/],
                cardMax=[dattrCasted.dassoc.associate.cardMax/]
    ))[dattrCasted.isVisibility/] Collection<[dattrCasted.select.name/]> [dattrCasted.name/];
[elseif (dattrCasted.type = dcsl::AttrType::Domain)]
    @DAttr(name = "[dattrCasted.name/]",attrType=[dattrCasted.type/])
[if (dattrCasted.dassoc <> null)]
    @DAssoc(ascname="[dattrCasted.dassoc.ascName/]",role="[dattrCasted.dassoc.role/]" ,
            asocType=[dattrCasted.dassoc.ascType/],endType=[dattrCasted.dassoc.endType/],
        associate=@Associate(attrType=[dattrCasted.select.name/].class,
                                 cardMin=[dattrCasted.dassoc.associate.cardMin/],
                                 cardMax=[dattrCasted.dassoc.associate.cardMax/]))
[/if]
    [dattrCasted.isVisibility/] [dattrCasted.select.name/] [dattrCasted.name/];
```

Fig. 6. Acceleo source code maps classes, properties, and methods in the domain model

In lines 4 and 5 of Algorithm 1, the **Rule 2** T will map module specifications, configuration, and user interface information from the UDM to Java annotations.

- *Name of module class Module:* (1) Each *Module* in the UDM will be converted to a Java class with the functions described in the model. (2) The name of the Java class will be derived from the name of the module.
- *View:* The *ViewDesc* annotation describes the module's user interface information.
- *Controller:* The *ControllerDesc* annotation (if any) describes the component's control information.
- *Model:* The *ModelDesc* annotation describes the domain class information the module is using.
- *AttributeDesc:* The *@AttributeDesc* annotation describes the attributes and roles in the module. Attributes map to attributes in the domain model class.
- *Edge:* The *CTree* annotation describes the hierarchical relationship between modules in the software and how the modules connect to each other.
- *Scope:* The *ScopeDesc* annotation describes the operating scope for each *Control*, *View*, and *Model* configuration.
- *Prop:* The *Prop* annotation provides additional information for *Model*, *View*, and *Controller* configuration.

Figure 7 shows a part of the Acceleo code that maps module specifications, configuration, and user interface information.

mccl_generate.mtl

```
[comment encoding = UTF-8 /]
[module mccl_generate('http://www.example.org/dcsl',
'http://www.eclipse.org/emf/2002/Ecore','http://www.example.org/mosa')]
[template public generateElement_mccl(m_module : Module)]
@ModuleDescriptor(
      name= Module[m_module.name/]
      modelDesc=@ModelDesc(
    model=[m_module.modelconfig.domainclass.name/].class
    ),
     viewDesc=@ViewDesc(
     formTitle="[m_module.viewconfig.formTitle/]"
    ),
    [if (m_module.controlconfig->notEmpty())]
    controllerDesc=@ControllerDesc(
        controller=Controller.class
        [if (m_module.controlconfig.prop->notEmpty())]
        ,props= {
          @PropertyDesc(name=PropertyName.controller_dataController_new,
        valueIsClass=ExecActivityCommand.class, valueType=Class.class, valueAsString=CommonConstants.NullString)
        }
        [/if]
    [/if]
  [if m_module.class.eClass().name->first() = 'ActivityClass']
), containmentTree=
    root=[m_module.name/].class
    [for (cTree : agl:: ActivtyGraph | m_module.containmentTree)]
    @CTree(
    [for (aGraph : agl:: ActivtyGraph | cTree)]
        [for (edge : agl:: Edge | aGraph.edge)]
         [if (edge.target.label = 'CustOrder' and (edge.source.nodeType = agl::NodeType::Null
    or edge.source.nodeType = agl::NodeType::Coordinator or edge.source.nodeType = agl::NodeType::Fork))]
                    parent=[edge.source.label/].class, child=[edge.target.label/].class,
)),
```

Fig. 7. Acceleo code that maps module specifications, configuration, and user interface information

In lines 6 to 9 of Algorithm 1, the **Rule 3 and 4** T will map nodes inside the *ActivityGraph* in the UDM to Java source code, ensuring consistency in the source code.

- *RootNode:* The *RootNode* will be transformed into a Java class representing the activity graph. The name of the Java class will be based on the *RootNode*.
- *Node:* The *ANode* annotation describes information about nodes in the activity graph.
- *ModuleAct:* The *MAct* annotation describes the state and actions of buttons in the activity graph controls, except buttons.
- *Coordinator:* Located in the same module as the activity class, determined using if/else statements

Figure 8 shows a part of the Acceleo map nodes inside the *ActivityGraph* in the UDM to Java source code, ensuring consistency in the source code.

In lines 10 and 11 of Algorithm 1, the **Rule 5** T will Map high-level software configuration specifications from the model to Java annotations conform to MOSA.

- *Software:* The *Software* tag in the UDM will be translated into a Java class with functions described in the model. The name of the Java class will be based on the software
- *Configuration:* Annotations such as *SystemDesc*, *Database*, and *Security* describe software information, supplemented by configuration files for all software artifacts.
- *List of modules unified AGL, DCCL, MCCL:* Utilize a for loop to incorporate all modules contained within.

Figure 9 illustrates a section of the Acceleo code that maps software configuration specifications to conform with

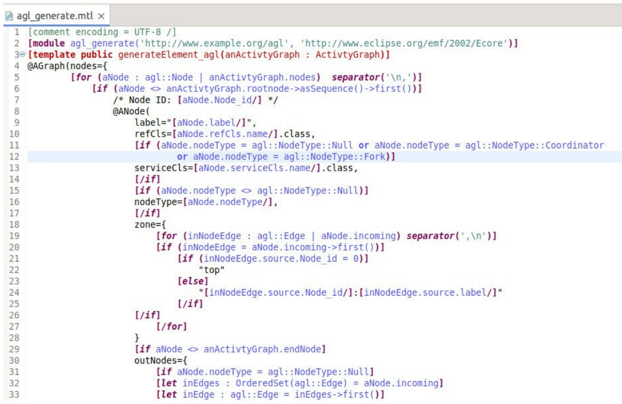

Fig. 8. Acceleo map nodes inside the *ActivityGraph* in the UDM to Java source code, ensuring consistency in the source code

the MOSA architecture and enhances automation for the JDA method.

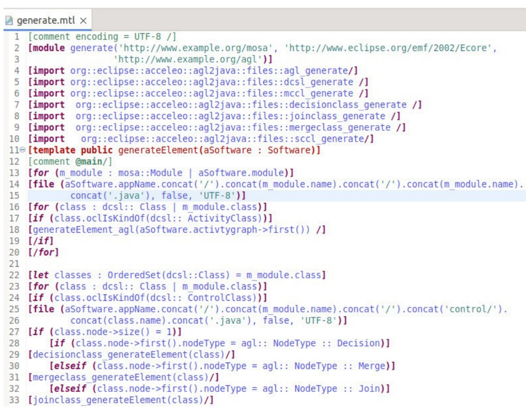

Fig. 9. Acceleo maps software configuration specifications

VI. Tool Support and Experiments

In this section, we present a case study of OrderMan, using the Order management system, to illustrate our UDML. The metamodel describing the abstract syntax of the UDML language is represented by the Figure 10 including three basic components: (1) The metamodel describes the AGL (behavior and structure of the domain model) structure. (2) The metamodel describing the structure of DCSL (composed of six metaconcepts: Class, Field, Method, Parameter, Annotation, and Property class, and a constraint) is the component to convert into the template configured unified model of the software as activity graphs. (3) The metamodel representing the structure of MCCL (Module Configuration Code Language) includes three configuration components: ModelConfig, ViewConfig, and ControllerConfig, defined by the ModuleConfig's metamodel.

Figure 11 shows the unified domain model on the left. On the right, it shows the resulting HandleOrder model. We then implement our method using a supporting tool built on JDA, a Java software framework. We use Acceleo

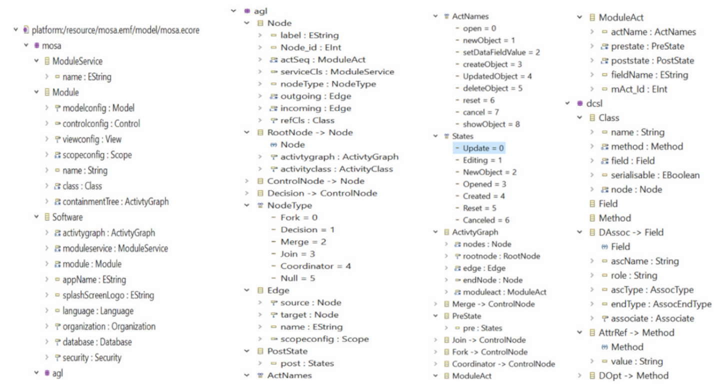

Fig. 10. meta-model of UCML

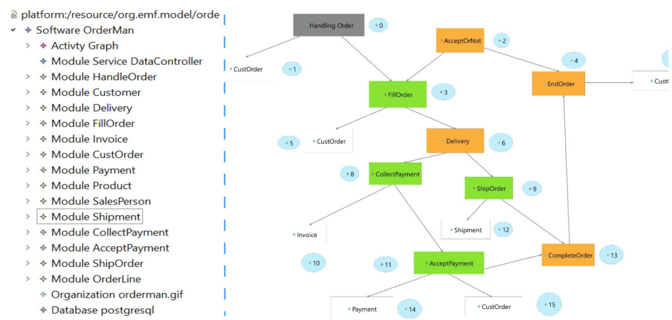

Fig. 11. tool suport use UDML with the method proposed

to perform a transformation from the MOSA model to Java code and implementation in JDA. Figure 12 shows the resulting software. The top left of the figure presents the Acceleo code, the bottom left presents the result of the generation, and the right side of the figure shows the GUI of the Orderman implementation in JDA, a Java software framework.

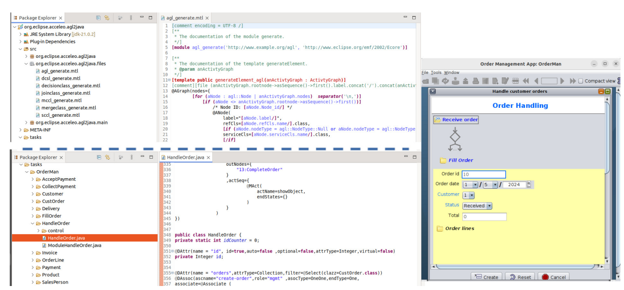

Fig. 12. Running example of OrderMan

VII. Conclusion and Future Work

In this paper, we introduce a technique to generate software artifact follow the MOSA architecture with a new domain model specification, using external DSL and ensuring consistency between external DSL and internal DSL using model transformation techniques.

Our approach involves a DSL UDML and providing a transformation-based method to automatically generate software artifacts from a metamodel using Acceleo. (1) UDML is a unified-domain modeling language AGL, DCSL and MCCL with the semantics of it allows us to precisely explain the meaning of unified domain model and provides a basis for transformations to automatically generate software artifacts from a UDM. (2) A tool support for our method on transforming and implementing software artifact generation for domain-driven design.

In the future, we plan to develop an Eclipse plug-in for our method from the use case model. We also intend to develop a technique for automatically transforming a specification use case into a complex software systems.

References

[1] D.-H. Dang, D. M. Le, V.-V. Le, Agl: Incorporating behavioral aspects into domain-driven design, Information and Software Technology 163 (2023) 107284. doi:https://doi.org/10.1016/j.infsof.2023.107284.

[2] D. M. Le, D.-H. Dang, V.-H. Nguyen, On domain driven design using annotation-based domain specific language, Computer Languages, Systems & Structures 54 (2018) 199–235.

[3] D. M. Le, D.-H. Dang, V.-H. Nguyen, Generative software module development for domain-driven design with annotation-based domain specific language, Information and Software Technology 120 (2020) 106239.

[4] E. Evans, Domain-driven design: tackling complexity in the heart of software, Addison-Wesley Professional, 2004.

[5] Apache-S.F, Apache Isis (2022).
URL `http://isis.apache.org/`

[6] Apache-S.F, Apache Causeway (2023).
URL `https://causeway.apache.org/`

[7] OpenXava (2022).
URL `http://openxava.org/`

[8] OMG, Uml 2.5.1 uml-unified modeling language (2017).

[9] M. Dumas Menjivar, A. Ter Hofstede, Uml activity diagrams as a workflow specification language, in: Proceedings of the 4th International Conference UML 2001 (The Unified Modeling Language: Modelling Languages, Concepts, and Tools), Springer, 2001, pp. 77–90.

[10] A. Kleppe, Software language engineering: creating domain-specific languages using metamodels, Pearson Education, 2008.

[11] T. Clark, A. Evans, S. Kent, Aspect-oriented metamodelling, The Computer Journal 46 (5) (2003) 566–577.

[12] O. A. Specification, Meta object facility (mof) 2.0 core specification.

[13] V. Vernon, Implementing domain-driven design, Addison-Wesley, 2013.

[14] D. M. Le, D.-H. Dang, V.-H. Nguyen, Generative software module development: A domain-driven design perspective, in: 2017 9th International Conference on Knowledge and Systems Engineering (KSE), IEEE, 2017, pp. 77–82.

[15] D. Karagiannis, H. C. Mayr, J. Mylopoulos, Domain-specific conceptual modeling, Springer, 2016.

[16] M. Leduc, On modularity and performances of external domain-specific language implementations, Ph.D. thesis, Université de Rennes (2019).

[17] R. Haesen, M. Snoeck, Implementing consistency management techniques for conceptual modeling, consistency problems in UML-Based software development (2005).

[18] G. Engels, R. Heckel, J. M. Küster, L. Groenewegen, Consistency-preserving model evolution through transformations, in: UML 2002—The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference Dresden, Germany, September 30–October 4, 2002 Proceedings 5, Springer, 2002, pp. 212–227.

[19] J. M. Küster, Consistency management of object-oriented behavioral models, Shaker, 2004.

[20] M. Fowler, D.-S. Languages, Addison-wesley professional (2010).

[21] S. Kelly, J.-P. Tolvanen, Domain-specific modeling: enabling full code generation, John Wiley & Sons, 2008.

[22] J. Warmer, A. Kleppe, Building a flexible software factory using partial domain specific models, in: 6th OOPSLA Workshop on Domain-Specific Modeling, DSM 2006, University of Jyvaskyla, 2006, pp. 15–22.

[23] A. Wąsowski, T. Berger, Domain-Specific Languages: Effective modeling, automation, and reuse, Springer, 2023.

[24] S. Kapferer, O. Zimmermann, Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling., in: MODELSWARD, 2020, pp. 299–306.

[25] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Synthesis lectures on software engineering 3 (1) (2017) 1–207.

[26] K. Czarnecki, Overview of generative software development, in: International workshop on unconventional programming paradigms, Springer, 2004, pp. 326–341.

[27] A. Van Deursen, E. Visser, J. Warmer, Model-driven software evolution: A research agenda, in: Proceedings 1st International Workshop on Model-Driven Software Evolution, MoDSE Nantes, France, 2007, pp. 41–49.

[28] H. J. Köhler, U. Nickel, J. Niere, A. Zündorf, Integrating uml diagrams for production control systems, in: Proceedings of the 22nd international conference on Software engineering, 2000, pp. 241–251.

[29] I. A. Niaz, J. Tanaka, An object-oriented approach to generate java code from uml statecharts, International Journal of Computer & Information Science 6 (2) (2005) 83–98.

[30] A. G. Parada, E. Siegert, L. B. De Brisolara, Generating java code from uml class and sequence diagrams, in: 2011 Brazilian Symposium on Computing System Engineering, IEEE, 2011, pp. 99–101.