On Automatic Generation of Executable Domain Models for Domain-Driven Design

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

Nghia-Trong Be
Department of Software Engineering
University of Engineering
and Technology
Vietnam National University, Hanoi
20021400@vnu.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 model of the problem domain, that is both a thorough grasp of the domain requirements and technically feasible to deploy. Current works, approaches and methodologies that transform a high-level specification (UML behavior models) into source code aim to simplify software development and bring the software to market quickly. However, transforming a high-level specification into an executable domain model for DDD is challenging and requires a precise specification of the domain model at high-level abstraction with UML Activity diagram and UML/OCL Class diagram. In this paper, we propose a technique to obtain a precise specification (an executable model in DCSL and AGL) of the domain model for DDD by a transformation from a specification at high-level abstraction (with UML Class and Activity diagrams). We develop a tool support for our method using Acceleo to automatically transform high-level specification into source code and executable models in DCSL and AGL based on jDomainApp, a Java software framework.**

***Index Terms*—DDD, UML/OCL, Model Transformation, Class Diagram, Activity Diagram**

I. Introduction

Domain-driven design (DDD) [1] aims to iteratively develop software around a realistic model of the problem domain, that is both a thorough grasp of the domain requirements and technically feasible to deploy. The Model-Driven Development (MDD) [2] approach moves the focus on models in software development and those models will be automatically generated into implementation code. In our recent work, we incorporated domain behaviors into a domain model using a composition of both structural and behavioral aspects of the domain, namely AGL [3]. In this research, we realize the difficulty of defining a precise specification at high-level abstraction (with UML Class and Activity diagrams) and transform it into a DCSL [4] and AGL specification (so-called AGL^+ specification) of the domain model for DDD to generate software artifacts.

The high-level specification (which consists of UML Activity and Class diagram) is one of the behavioral models used for modeling the global behavior of systems, it models both data and control flow in the system. Meta-modeling for DSL [5] is used 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 [6] involves the construction of an object-oriented model of the abstract syntax optionally, the concrete notation and semantics of the target language [7]. The main aim of the proposed high-level specification is to automatically generate the AGL^+ specification.

Within our approach, we aim to transform model to text as we input a specification at high-level abstraction as the input domain model to transform it in to source code (AGL^+ specification) for the usage of the DDD methodology of the host programming language (such as Java). In this work, we define a domain model (a high-level specification abstraction) and a technique for transforming the rules to AGL^+ specification. To demonstrate our method, we use Acceleo [8] to transform a domain model to AGL^+ and use a Java framework [9] called jDomainApp to generate software through a case study to show how the AGL^+ specification can be applied to real-world software.

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

- A technique to obtain a precise specification (an executable model in DCSL and AGL) of the domain model for DDD by a transformation from a specification at high-level abstraction (with UML Class and Activity diagrams).
- A tool support for our method on transforming and implementing software artifact generation for domain-driven design.

The rest of the paper is organized as follows: Section II introduces the motivating example and background for our work. Section III overviews our proposed approach. Section IV presents the transformation of UML Activity diagrams into AGL and the rules (templates in Acceleo to automate the generation process). Section VI discusses related works. The paper is closed with a conclusion and

future works.

II. Motivating Example and Background

A. Motivating Example

We use the order management domain OrderMan [6, p. 396] as our motivating example. The OrderMan is a complex software model, which is adapted from the OMG/UML specification with an Activity diagram and a Class diagram together with OCL constraints. Figure 1 shows an input domain model for OrderMan. As a specification, a full input model contains various kinds of `ControlNodes`. A `DecisionNode` after ReceivedOrder illustrates a branching based on two possible conditions: OrderRejected or OrderAccepted. FillOrder is followed by a `ForkNode` that passes control to both SendInvoice and ShipOrder nodes. A `JoinNode` indicates that the control will be passed to a `MergeNode` when both ShipOrder and AcceptPayment are completed. As the `MergeNode` will simply pass the token along, the EndOrder node will be executed.

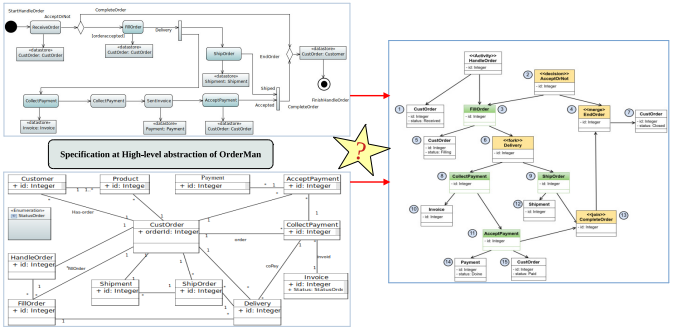


Fig. 1. Motivating Example

Our research problem, we deal with the domain problem using a four-step solution. Firstly, we specify a precise UML Activity diagram of the OrderMan and the guards for coordinate nodes. Secondly, we combine the UML Class diagram with the OCL constraints. Within the DDD approach [4] this domain model would be represented in DCSL. With these two steps, we employ a technique to obtain a precise specification at high-level abstraction (with UML Class and Activity diagrams) which is shown in the right Figure 1, in which we created the input domain model then moved on to the next step. Thirdly, there is a major challenge in this step, which is in the process of transforming the domain model to AGL and DCSL specification, then combine them into a AGL^+ to obtain activity graph configuration of the unified model which consists of activity graph (AGraph), activity node (ANode), and module action (MAct) of AGL. We use annotations (@AGraph, @ANode, @MAct) and DCSL in our recent work [3]. Finally we execute AGL^+ of the domain model for DDD using the framework: jDomainApp [9].

A Class diagram [6] describes the structure of a system by showing the classes of the systems, their attributes, and the relationships among the classes. For UML class diagrams, OCL is the language of choice for defining constraints going beyond simple multiplicity and type constraints in Figure 2.

The UML Activity Diagram is often used to model the overall system behavior and to show the entire business process flow. Many approaches and methodologies have been proposed for the automatic source code generation from the UML Activity Diagram [10]–[12]. In [11] the authors used a modified DFS (Depth-First Search) algorithm to generate a set of paths from the Activity diagram. For example, Figure 2 shows a newly created instance of the class `HandleOrder` which is received by the action `ReceiveOrder` (the operation `initHandleOrder`() of the class `HandleOrder` is invoked). Subsequently, if the guard condition (`Self.rejectOrder`()) of the decision node is false (indicating that the order is not rejected), the flow proceeds to the next step: the action `FillOrder` (`Self.fillOrder`()). After that, the fork node splits the path of the control flow into two parallel tasks. If, however, the order is rejected, the flow directly proceeds to the action `EndOrder`.

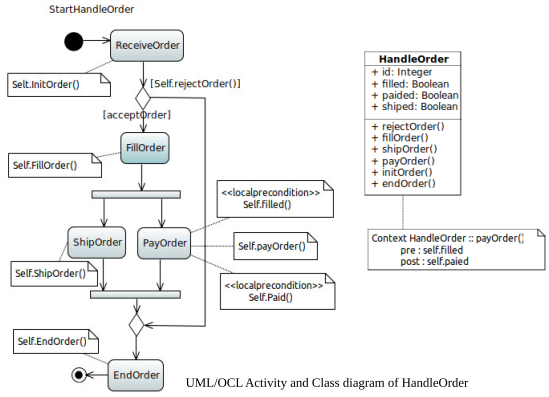


Fig. 2. The UML/OCL Activity and Class diagram of HandleOrder of OrderMan

B. Specifying Precisely Domain Models for a Domain-Driven Design

Domain-Driven Design (DDD)

Domain-Driven Design (DDD) [1], [13] aims to develop complex software. The DDD methodologies captures the domain requirements and is technically feasible for implementation. In practice, complex software requires a scalable architecture and a highly productive software development method. Within the DDD approach [1], the domain model tends to be the heart of software, the domain model would satisfy the domain requirements that are expressed in a ubiquitous language. This language is defined for stakeholders, including the domain experts and developers, in an iterative and agile process of eliciting the domain requirements. This work uses DDD to refer specifically to object-oriented and to construct DDD to generate software artifacts from a domain model. The

domain model is feasible for implementation in a host OOPL [9].

A DCSL and AGL specification

Annotation-Based Domain Specific Language (aDSL) is coined in [14] as an attempt to formalise the notion of fragmentary, internal DSL [15] for the use of annotation to define DSLs. An aDSL is defined based on a set of meta-concepts that are common to two popular host OOPLs, the Java [16] and the C#. **Domain class specification language** (**DCSL**) [4] is a horizontal aDSL developed by us to express domain models.

In this work, we use feature of DCSL, which uses its meta-concepts model the domain-specific terms composed of the core OOPL meta-concepts and constraints. More specifically, the meta-concepts includes: **Domain Class**, **Domain Field**, **Associative Field** and **Domain Method** .

In our previous work [3], we proposed Activity Graph Language (AGL) to incorporate domain behaviors into a domain model: AGL is defined to represent the domain behaviors for such incorporation. Each domain behavior, described at high level using a UML Activity diagram and domain-model based statements, is translated into a specification with two parts: (1) a part of the unified class model with new activity classes, and (2) the activity graph logic of the input activity and the mappings to connect the activity with the unified class model.

To create software artifacts following the DDD approach, we must combine DCSL and AGL specification (namely AGL^+ specification), which can be use to produce in a Java program.

The AGL+ specification of OrderMan, resulting in a generated software

The activity diagram specified by the AGL represents the `HandleOrder` class. Using the annotation mechanism in Java, the HandleOrder object can be treated as an AGraph object, allowing it to represent and manage the Activity diagram. The AGraph object enables to handle each of its ANodes. Listing 1 illustrates the annotation used to express the ANode w.r.t node 14 the AGL specification of the `HandleOrder` activity class. The class diagram specified by the DCSL for the class `HandleOrder` of OrderMan, which is written in Java. Listing 2 provides an example class `HandleOrder` which is specified with `DClass.serialisable = false`. The domain field name is mutable with `DAttr.mutable = true`.

Listing 1. The activity diagram `HandleOrder` in Java AGraph

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

Listing 2. DCSL specification of `HandleOrder` written in Java

```
/** DCSL specification of \clazz{HandleOrder} written in Java*/
@DClass(serialisable=false, singleton=true)
public class HandleOrder {
  @DAttr(name = "id", id = true, auto = true, type = Type.
       Integer, length = 5,
  optional = false, mutable = false)
  private int id;
  private static int idCounter = 0;
  ...}
```

C. Research Questions

To achieve this goal, we face the following primary challenge that motivate our work:

How can we obtain an AGL^+ specification by a model transformation from a specification of the domain model at high-level abstraction with UML Activity diagram and UML/OCL Class diagram?

III. Overview of Our Approach

This section explains our basic idea on transforming a high-level specification to an executable domain model for domain-driven design. Figure 3 overviews our proposed

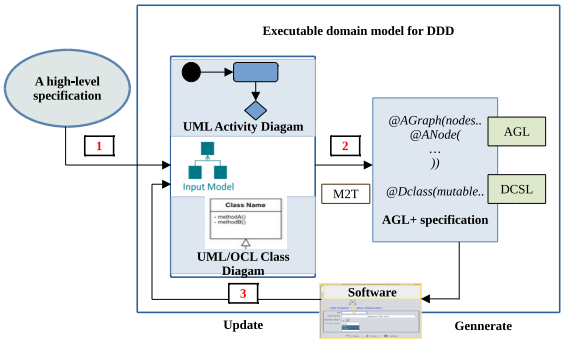


Fig. 3. Overview of Our Approach

technique. This technique conceptually consists of iteratively performing three steps.

Step 1 (The construct domain model takes input): We take a specification with UML/OCL Class Diagram (CD) and Activity Diagram (AD) as input. To achieve target of AGL^+ specification, we transform the CD input to the DCSL specification and the AD input to the AGL specification. A detailed technical specification on how to generate AGL^+ specification from high-level specification AD and CD is in Section IV. Here, we consider the AGL^+ specification as an extended domain model in a module-based software architecture [9]. It is expressed in AGL and DCSL as an extension of the conventional DDD's domain model [1] with the executable domain model.

Step 2 (The AD2AGL transformation): This step transforms the input AD specification into an AGL specification and then composes it with DCSL specification, which is implemented in DDD as a GUI- and module-based software. This software is presented to the domain expert in order to get feedback.

Step 3 (The update and the cycle): If there is feedback, then the input domain model will be updated and the

cycle continues. If, on the other hand, the domain expert is satisfied with the models; then the cycle ends.

IV. Transforming ADs to the AGL Specification

A. Algorithm AD2AGL

This section presents an algorithm to generate an AGL^+ specification of the domain model from an UML Activity and Class diagram. We call it the AD2AGL algorithm.

Algorithm 1 AD2AGL takes as input AD (the UML Activity diagram that captures the system behavior) and CD (the UML Class diagram that specifies the domain model), and produces as output DM (an AGL+ specification of the domain model with annotated and realized in Java; e.g. @AGraph, @ANode, @MAct, @DClass, @DAttr, etc.).

Algorithm 1 Generating an AGL specification from an UML Activity diagram

Require:

- *AD*: the UML Activity diagram to capture the system behavior.
- *CD*: the UML class diagram to specify the domain model.

Ensure:

- *DM*: an AGL^+ specification of the domain model

1: **Initialization:** $AG \Leftarrow$ the activity graph w.r.t. *AD*; $curNode \Leftarrow$ the initial nodel of *AG*
2: **while** *curNode* is not the final node of *AG* **do**
3: **if** *curNode.next* is a Decision node **then**
4: $genDecisionNode(curNode, DM)$
5: **else if** *curNode.next* is a Sequential node **then**
6: $genSequentialNode(curNode, DM)$
7: **else if** *curNode.next* is a Fork node **then**
8: $genForkNode(curNode, DM)$
9: **else if** *curNode.next* is a Merge node **then**
10: $genMergeNode(curNode, DM)$
11: **else if** *curNode.next* is a Join node **then**
12: $genJoinNode(curNode, DM)$
13: **for each** $class \in CD$ **do**
14: $genDCSL(class, DM)$

First, we initialize an empty DM to contain an AGL+ specification of the domain model and point to the initial node of AD.

Second, it uses the Depth-First Search algorithm on the AD to visit all nodes in the activity graph. An AGL specification will be added whenever a new action node is visited. As per UML [6], an activity diagram is considered a graph. An Activity Graph (AG) [11] is a hextuple containing nodes N, edges E, events *eve*, guard conditions *gua*, local variables *var*, and set of objects *obj*. A node can be of two types: *ActionNode* and *ControlNode*. The *ActionNode* includes *action node, acceptEvent node,* and *sendSignal node*. The *ControlNode* includes *initial node, final node, sequential node, decision node, merge node, fork* and *join node* and for more information, we refer the reader to Appendix A of the technical report [17] accompanying this paper.

Third, it checks each node in the AD, including all action nodes and control nodes, and calls sub-functions for the respective cases to generate *AGL* specification. When visiting the current node and identifying the next node from the current node, it checks the next node and calls the functions for generating *AGL* with input parameters corresponding to the current node ($curNode$) and updated the *AGL* specification of the domain model (DM). The functions are: *genSequentialNode(curNode,DM)*, *genDecisionNode(curNode,DM)*, *genForkNode(curNode,DM)*, *genMergeNode(curNode,DM)* and *genJoinNode(curNode,DM)*. The function adds an AGL specification according to the annotation-based textual concrete syntax model for AGL that is defined in [3].

Fourth, it visits the next node and repeats step third until all nodes have been visited (meet *finalNode*).

Fifth, generate DCSL from CD by traversing all classes on CD and updating the DCSL specification in the domain model (DM).

we will explain the functions for the sequential, decision, fork, merge and join node in the *genSequentialNode(curNode,DM)*, *genDecisionNode(curNode,DM)*, *genForkNode(curNode,DM)*, *genMergeNode(curNode,DM)* and *genJoinNode(curNode,DM)*) and their node specifications in AD and the AGL specification updated to DM.

The function *genDCSL(class,DM)* generates DCSL from CD (DCLS specification according to the annotation-based approach is defined in [4]) by traversing all classes on CD and updating the DCSL specification in the domain model (DM).

Algorithm generate Sequential node

In the function *genSequentialNode(curNode,DM)*, which takes the input $curNode$ - the current node, which has a next action node named e_s in AD, represented in Figure 4.

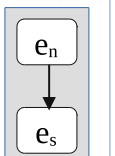
Sequential specification in Ad	**Desciption in algorithm: *callsequentialAGL(e_n:ActivityNode)***
sequential pattern (n>1) 	the name of current node (action node): e_n
	the name of next node of the e_n: e_s
	the name of the incoming edge of the current node "as"

Fig. 4. The sequential node specification in AD

- The function $genAGLcurNode(curNode, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls=$curNode$, seviceCls=DataController, outClses=[$curNode.Next$]

Algorithm 2 Generating an AGL specification from sequential node in AD

Require:

- $curNode$: the current node in AD
- DM: an AGL+ specification of the sub-domain model

Ensure:

- DM: an AGL+ specification of the domain model

1: $e_s \Leftarrow$ the $curNode.next$
2: $genAGLcurNode(curNode, DM)$
3: $genAGLEs(e_s, DM)$

actSeq=[MAct{actName=newObject, pstStates=[Created]}] init=true}

- The function $genAGLEs(e_s, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls=e_s, seviceCls=DataController, actSeq=[MAct{actName=newObject, pstStates=[NewObject]}, MAct{actName=setDataFieldValue, fieldNames=["as"] pstStates=[Created]}]}

Algorithm generate Decision node

In the function *genDecisionNode(curNode,DM)*, which takes the input $curNode$ - the current node, which has a next decision node named e_d in AD, represented in Figure 5.

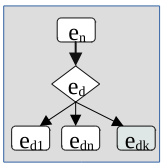
Decision specification in Ad	Update to DM an AGL specification
decision pattern (if e_{dk} is present then $n>1$ else $n>2$) 	**@Anode**{refCls= e_n, outClses=[e_d], …} **@ANode**{refCls=e_d , nodeType = Decision, outClses = [e_{d1} , e_{dn}, e_{dk}] ,…} **@ANode**{refCls = e_{d1} , fieldNames = ["ad"], ..} **@ANode**{refCls = e_{dn} , @MAct{actName =…} **@ANode**{refCls = e_{dk} , nodeType = k- specifies the control node type, outClses =k_{out} - specifies the array of output e_{dk} }

Fig. 5. The decision node specification in AD

- The function $genAGLcurNode(curNode, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls=$curNode$, seviceCls=DataController, outClses=[e_d], actSeq = [@MAct{actName = newObject, pstStates = [Created]}], init = true}
- The function $genAGLEd(e_d, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls=e_d, nodeType = Decision seviceCls = DataController, outClses = [all next node of e_d: e_{d1}, e_{dn} and data store node e_{dk}]}
- The function $genAGLEdk(e_{di}, DM)$ generates AGL that adds the following content to the DM:

Algorithm 3 Generating an AGL specification from decision node in AD

Require:

- $curNode$: the current node in AD
- DM: an AGL+ specification of the sub-domain model

Ensure:

- DM: an AGL+ specification of the domain model

1: $e_d \Leftarrow$ the $curNode.next$
2: $genAGLcurNode(curNode, DM)$
3: $genAGLEd(e_d, DM)$
4: **for all** $e_d.edge(any)$ **do**
5: $\quad e_{di} \Leftarrow e_d.next$
6: $\quad$ **if** e_{di} is the data store node **then**
7: $\quad\quad genAGLEdk(e_{di}, DM)$
8: $\quad$ **else**
9: $\quad\quad genAGLEdi(e_{di}, DM)$

@ANode{refCls = e_{dk}, nodeType = k- specifies the control node type, outClses =k_{out} - specifies the array of output e_{dk}}

- The function $genAGLEdi(e_{di}, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls = e_{d1}, serviceCls = DataController, actSeq = [@MAct{actName = newObject, pstStates = [NewObject]}, @MAct{actName = setDataFieldValues, fieldNames = ["ad"], pstStates = [Created]}] } @ANode{refCls = e_{dn}, serviceCls = DataController, actSeq = [@MAct{actName = newObject, pstStates = [Newobject]}, @MAct{actName = setDataFieldValues, fieldNames = ["ad"], pstStates = [Created]}]}

Algorithm generate Fork node

In the function *genForkNode(curNode,DM)*, which takes the input $curNode$ the current node, which has a next fork node named e_f in AD, represented in Figure 6.

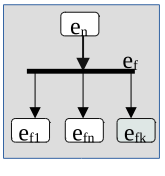
Fork specification in Ad	Desciption in algorithm: ***callforkAGL(e_n:ActivityNode)***
forked pattern (if e_{fk} is present then $n>1$ else $n>2$) 	the name of the current node (action node): e_n
	the name of the fork node: e_f
	the name of the nodes thats fork node links to: $e_{f1,}$ e_{fn} and e_{fk} (e_{fk} is the data store node)

Fig. 6. The fork node specification in AD

- The function $genAGLcurNode(curNode, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls=$curNode$,

Algorithm 4 Generating an AGL specification from fork node in AD

Require:

- $curNode$: the current node in AD
- DM: an AGL+ specification of the sub-domain model

Ensure:

- DM: an AGL+ specification of the domain model

1: $e_f \Leftarrow$ the $curNode.next$
2: $genAGLcurNode(curNode, DM)$
3: $genAGLEf(e_f, DM)$
4: **for all** $e_f.edge(any)$ **do**
5: $e_{fi} \Leftarrow e_f.next$
6: **if** e_{fi} is the data store node **then**
7: $genAGLEfk(e_{fi}, DM)$
8: **else**
9: $genAGLEfi(e_{fi}, DM)$

seviceCls=DataController, outClses=[e_f] actSeq=[MAct{actName=newObject, pstStates=[Created]}] init=true}

- The function $genAGLEf(e_f, DM)$ generates AGL that adds the following content to the DM: @ANodeANode{refCls=e_f, nodeType=Fork seviceCls=DataController, outClses=[all the next node of e_f: e_{f1}, e_{fn} and data store node e_{fk}]}
- The function $genAGLEfk(e_{fk}, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls= e_{fk},nodeType=k- specifies the control node type, outClses=k_{out} - specifies the array of output e_{dk} }
- The function $genAGLEfi(e_{fi}, DM)$ generates AGL that adds the following content to the DM: @ANode{refCls= name of node e_{f1}, serviceCls=DataController actSeq=[MAct{actName=newObject, pstStates=[NewObject]}, MAct{actName=setDataFieldValues, fieldNames=["af"] pstStates=[Created]}] } @ANode{refCls= e_{fn},serviceCls=DataController actSeq=[MAct{actName=newObject, pstStates=[Newobject]}, MAct{actName=setDataFieldValues, fieldNames=["af"] pstStates=[Created]}]}

B. Rules (templates in Acceleo)

To implement the code generator, the Model-to-Text transformation (M2T) is needed to generate the AGL^+. We define the rules (templates in Acceleo) to transform from the high-level specification AD and CD of OrderMan to AGL^+ in Listing 3.

Listing 3. The template transformation AD to AGL

```
[template public generateElementAGL(aModel : Model)]
[file (aModel.name.concat('.java'), false, 'UTF-8')]
/**AGL specification*/
@Agraph(nodes={
    [for ( e: PackageableElement| packagedElement)]
    [if(e.oclIsKindOf(Activity))]
    [let aActivity : Activity = e.oclAsType(Activity)]
    [for (aActivityNode : ActivityNode | aActivity.node)]
    [let nextNode : ActivityNode =    callAction(
         aActivityNode)]
    [if (nextNode.oclIsKindOf(OpaqueAction))]
    [callSequentialAGL(aActivityNode)/]   [/if]
    [if (nextNode.oclIsKindOf(DecisionNode))]
    [callDecisionAGL(aActivityNode)/]   [/if]
    [if (nextNode.oclIsKindOf(ForkNode))]
    [callForkAGL(aActivityNode)/]     [/if]
    [if (nextNode.oclIsKindOf(MergeNode))]
    [callMergeAGL(aActivityNode)/]      [/if]
    [if (nextNode.oclIsKindOf(JoinNode))]
    [callJoinAGL(aActivityNode)/]     [/if]
    [/let]  [/for]  [/let]  [/if] [/for]})
  /** DCSL specification */
    genDCSL(model:Model)
```

We define the rules (templates in Acceleo) for the transformation of each node in the UML AD. we will illustrate the *Decisional* node, for more information on the other nodes (*Sequential, Forked, Joined and Merged*), please refer to Appendix B in this paper. The template in Acceleo to transform a *Decisional* node in AD, with the function *genDecisionNode(curNode)*, is shown in Listing 4.

Listing 4. The template transformation Decision node to AGL

```
  /** Generated AGL specification from decision node*/
[template public genDecisionNode(enNode : ActivityNode)]
  [let edNode : ActivityNode = enNode.outgoing->any(true).
       target]
    @ANode{refCls=[enNode.name/],serviceCls=DataController,
         outClses=[edNode.name/],
      actSeq=[genMActcreateObject()/], init=true}
    @ANode{refCls=[edNode.name/], nodeType=Decision,
         outClses=
      [for (aNode : ActivityNode | edNode.outgoing->any(true
           ).target)]
        [aNode.name/]   [/for]
      [for (aNode : ActivityNode | edNode.outgoing->any(true
           ).target)]
      [if (aNode.oclIsKindOf(DataStoreNode))]
      @ANode{refCls=[aNode.name/], nodeType=[aNode.
           oclIsTypeOf(DecisionNode)/], outClses=[callAction
           (aNode)/]}
      [else]
      @ANode{refCls=[aNode.name/], serviceCls=DataController
           ,
        actSeq= [genMActnewObject()/]
        [genMActsetDataFieldValuesAndCreate()/]
      [/if]   [/for]    [/let]
  [/template]
```

To translate the OrderMan problem from UML/OCL Class diagram to DCSL specification using the function *genDCSL(model:Model)*, we design an input model that consists of classes, properties, associations and OCLs in CD, as shown in Listing C.

Listing 5. The template transformation CD to DCSL

```
  /** Transformation Class diagram to DCSL specification*/
  [template public genDCSL(model : Model)]
  public class HandleOrder{
    [for ( e: PackageableElement| packagedElement)]
    [if(e.oclIsKindOf(Class))]
    [let aClass : Class = e.oclAsType(Class)]
    [comment]-------------------Class HandleOrder
         ------------------------[/comment]
    [if aClass.name='HandleOrder']
    public class [aClass.name.toUpperFirst()/] {
      [genDClass('false', 'true')/]
      [for (aAtribute : Property | aClass.ownedAttribute)]
      [if (aAtribute.name='id')]
      [genDAttrID('id','true', 'true', 'Type.Integer', 5, '
           false', 'false')/]
      private int id;
      .... }
```

We use query templates to automatically generate **@MAct, @DClass**. In Listing 6, the **@MAct** and

@DClass are generated code according to the query template.

Listing 6. The template generate the *@MAct* and *@DClass*

```
  /**actName=newObject,pstStates={Created}*/
[query public genMActcreateObject() :
Set(String)='@MAct(actName=newObject,pstStates=[Created])'/]
/** $@$DClass(serialisable=true, singleton = true)*/
[query public genDClass(serialisable: String,singleton:
      String) :
Set(String)='@DClass(serialisable='.concat(serialisable).
      concat(',singleton=').concat(singleton).concat(')')
/]
```

V. Tool Support

In this section, we present a case study of order products management OrderMan to illustrate our technique of transforming the high-level specification AD of OrderMan to AGL, using Acceleo. Acceleo provides powerful tooling such as an editor with syntax highlighting, error detection, code completion, refactoring, debugger, profiler, and a traceability API that allows tracing model elements to the generated code and vice versa [8]. Figure 7 shows the AD of OrderMan at the top, which is the input model of our method. The bottom left shows the Acceleo code (template) that performs the transformation to the **HandleOrder** code, the output of our method is the AGL specification on the bottom right.

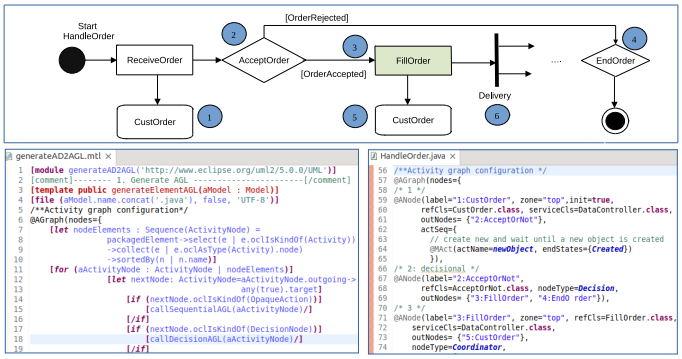


Fig. 7. Using Acceleo transformation AD of OrderMan to AGL, HandleOrder realized in Java

We also use Acceleo to transform the high-level UML/OCL Class diagram to DCSL specification resulting in Figure 8. We refer the reader to Appendix C in this paper.

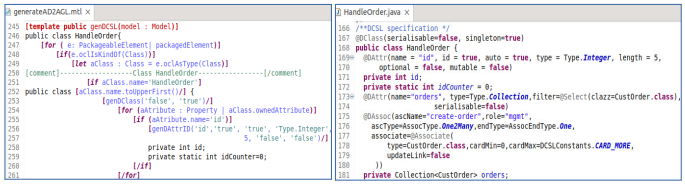


Fig. 8. Using Acceleo transformation CD of OrderMan to DCSL

Next, we compose the AGL with the DCSL to form the complete AGL^+ specification, which is used as input to automatically generate and implement our method. We demonstrate the implementation of our method, using a supporting tool built on JDomainApp, a Java software framework.

Figure 9 demonstrates the implementation of our method, using a supporting tool built on JDomainApp, a Java software framework.

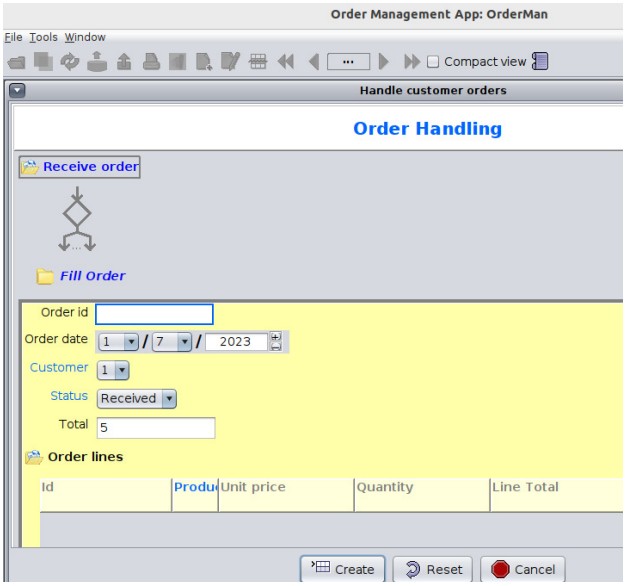


Fig. 9. The GUI of the Orderman software generated by the tool.

VI. Related Work

We position our work in the automatic generation AGL^+ specification by a model transformation from a specification of the domain model at high-level abstraction with UML AD and UML/OCL CD and executable domain model for DDD. Existing proposals for converting the behavioral models to the implementation code [10], such as the algorithm for code generation from activity and sequence diagrams [11], the authors used a modified DFS algorithm to generate a set of paths from the AD. Another proposal is the solution for automating transformation from CIM (Computing Independent Model) level business-based to the PIM (Platform Independent Model) level web-based [12].

Some methodologies aim to simplify software development by transforming a high-level specification (UML behavior models) into source code, the work [3] defines the abstract and concrete syntax of the AGL. However, the authors manually code from behavioral patterns (*Sequential Pattern, Decisional Pattern, Forked Pattern, Joined Pattern and Merged Pattern*) without automatic output generation of AGL^+ specification by a model transformation from a specification of the domain model at high-level abstraction with UML AD and UML/OCL CD.

To execute domain model for DDD, Evans proposed methodologies in the book [1] that capture the do-

main requirements and are technically feasible for implementation. However, he did not explicitly mention behavioral modeling as an element of the DDD method. The work [4] proposes an annotation-based DSL named DCSL for designing the domain model, and the work [9] proposes a generative software module development method for DDD, with a module-based software architecture and a generative technique for module configuration. The authors of these works have not yet proposed to automatically generate DCSL specification from at high-level abstraction (with UML Class and Activity diagrams). In this paper, we look at generating DCSL specification at high-level abstract and creating software artifacts following the DDD approach from combining DCSL and AGL specification into AGL^+ specification automatically.

VII. Conclusion and future work

In this paper, we have introduced a technique to transform a specification at high-level abstraction (with UML Class and Activity diagrams) into a precise AGL^+ specification of the domain model for DDD and a tool support, which is part of a Java framework that supports our technique and demonstrates the effectiveness of AGL^+ in designing real-world software. We find limitations of the method (the points would be part of our future work):

- The AD specification as combination of domain behavior patterns.
- The DCSL specification is semi-automatically transformed and manually written in Java: it should be obtained by a method that completely transforms the input UML/OCL Class diagram (or a DCSL model in graphical form)
- The composition of DCSL and AGL is manually performed.

In the future, we plan to develop an Eclipse plug-in for our method and create fully AGL^+ depending on the requirements specification. We also intend to develop a technique for automatically transforming at high-level requirements specification (use case) into a complex software systems.

Acknowledgments

We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

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

[2] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[3] D.-H. Dang, D. M. Le, and V.-V. Le, "Agl: Incorporating behavioral aspects into domain-driven design," *Information and Software Technology*, vol. 163, p. 107284, 2023.

[4] D. M. Le, D.-H. Dang, and V.-H. Nguyen, "On domain driven design using annotation-based domain specific language," *Computer Languages, Systems & Structures*, vol. 54, pp. 199–235, 2018.

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

[6] OMG, "Uml 2.5.1 uml-unified modeling language," 2017.

[7] T. Clark, A. Evans, and S. Kent, "Aspect-oriented metamodelling," *The Computer Journal*, vol. 46, no. 5, pp. 566–577, 2003.

[8] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

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

[10] E. Sunitha and P. Samuel, "Translation of behavioral models to source code," in *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)*, 2012, pp. 598–603.

[11] S. E. Viswanathan and P. Samuel, "Automatic code generation using unified modeling language activity and sequence models," *Iet Software*, vol. 10, no. 6, pp. 164–172, 2016.

[12] Y. Rhazali, Y. Hadi, I. Chana, M. Lahmer, and A. Rhattoy, "A model transformation in model driven architecture from business model to web model," *IAENG International Journal of Computer Science*, vol. 45, no. 1, pp. 104–117, 2018.

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

[14] M. Nosál, M. Sulír, and J. Juhár, "Language composition using source code annotations," *Computer Science and Information Systems*, vol. 13, no. 3, pp. 707–729, 2016.

[15] M. Fowler and D.-S. Languages, "Addison-wesley professional," 2010.

[16] J. Gosling, "The java language specification: Java se 8 edition," *(No Title)*, 2015.

[17] V.-V. Le, N.-T. Be, and D.-H. Dang, "On Transforming a High-Level Specification to an Executable Domain Model for Domain-Driven Design," VNU University of Engineering and Technology, Vietnam, Tech. Rep., 2023. [Online]. Available: https://tinyurl.com/AD2AGLTechnical

Appendix

A. Mapping the formalisation elements to metamodel

Element in metamodel	Components in formalisation
activity graph	AG
activityNode	N
activityEdge	E
events	eve
guard condition	gua
variable	var
objects	obj
Visit the next node	*callAction()*
Executable Node	*callsequentialAGL(), callforkAGL() calldecisionAGL(), callmergeAGL() and calljoinAGL()*

B. Template in Acceleo transformation sequential, fork, merge and join node in AD to AGL

The templates in Acceleo to transform *Sequential, Fork, Merge and Join* node in AD.

Template in Acceleo transformation sequential node in AD to AGL

Listing 7. Rules: transform sequential node in AD to AGL

```
/**visit next node*/
[template public callAction(currentNode : ActivityNode)]
[currentNode.outgoing->any(true).target/]
[/template]
/** Generate AGL specification for sequential node*/
[template public callSequentialAGL(esNode : ActivityNode
    )]
[let enNode : ActivityNode = callAction(esNode)]
@ANode{refCls=[esNode.name/], serviceCls=DataController,
      outClses=[enNode.name/],
  actSeq= [genMActcreateObject()/]}
@ANode{refCls=[enNode.name/], serviceCls=DataController,
  actSeq= [genMActnewObject()/].
  [genMActsetDataFieldValuesAndCreate()/]
  [/let]
  [/template]
```

We use the template to visit the next node in Listing 8.

Listing 8. Rules: visit the node in AD

```
/**visit next node*/
[template public callAction(currentNode : ActivityNode
    )]
[currentNode.outgoing->any(true).target/]
[/template]
```

Template in Acceleo transformation fork node in AD to AGL

Listing 9. Rules: transform fork node in AD to AGL

```
[template public callForkAGL(enNode : ActivityNode)]
  /** Generate AGL specification for Fork node*/
[let efNode : ActivityNode = enNode.outgoing->any(true).
    target]
@ANode{refCls=[enNode.name/],serviceCls=DataController,
    outClses=[efNode.name/],
  actSeq=[genMActcreateObject()/], init=true}
@ANode{refCls=[efNode.name/], nodeType=Fork, seviceCls=
    DataController, outClses=
  [for (aNode : ActivityNode | efNode.outgoing->any(true).
      target)]
  [aNode.name/]
  [/for]
  [for (aNode : ActivityNode | efNode.outgoing->any(true).
      target)]
  [if (aNode.oclIsKindOf(DataStoreNode))]
  @ANode{refCls=[aNode.name/], nodeType=[aNode.oclIsTypeOf(
      ForkNode)/], outClses=[aNode.outgoing->any(true).
      target.name/]}
  [else]
  @ANode{refCls=[aNode.name/], serviceCls=DataController,
    actSeq= [genMActnewObject()/]
    [genMActsetDataFieldValuesAndCreate()/]
    [/if]
    [/for]
    [/let]

    [/template]
```

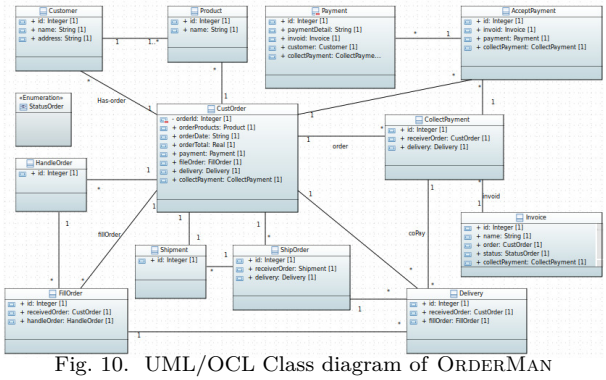


Fig. 10. UML/OCL Class diagram of OrderMan

Template in Acceleo transformation merge node in AD to AGL

Listing 10. Rules: transform merge node in AD to AGL

```
-------------
[/template]
```

Template in Acceleo transformation join node in AD to AGL

Listing 11. Rules: transform join node in AD to AGL

```
-------------
[/template]
```

C. Transformation UML/OCL Class diagram to DCSL specification

To translate the OrderMan problem from UML/OCL Class diagram to DCSL specification in Listing C, we design an input model that consists of Classes, properties, associations and OCLs in Figure 10.

```
/** Transformation Class diagram to DCSL specification*/
[template public genDCSL(model : Model)]
public class HandleOrder{
  [for ( e: PackageableElement| packagedElement)]
  [if(e.oclIsKindOf(Class))]
  [let aClass : Class = e.oclAsType(Class)]
  [comment]--------------------Class HandleOrder
       -------------------------[/comment]
  [if aClass.name='HandleOrder']
  public class [aClass.name.toUpperFirst()/] {
    [genDClass('false', 'true')/]
    [for (aAtribute : Property | aClass.ownedAttribute)]
    [if (aAtribute.name='id')]
    [genDAttrID('id','true', 'true', 'Type.Integer', 5, '
        false', 'false')/]
    private int id;
    private static int idCounter=0;
    [/if]
    [/for]
    public [aClass.name/](Integer id)
    { this.id=nextID(id);}
    //for use by object form
    public [aClass.name/]()
    { this(null);}
    public int GetId()
    { return id;}
    private static int nextID(Integer currID) {
      if (currID == null) { // generate one
        idCounter++;
        return idCounter;
      } else { // update
        int num;
        num = currID.intValue();
        if (num > idCounter) {
          idCounter=num;
        }
        return currID;
```

```
    }
  }
  [/if]

  [/let]
  [/if]
  [comment]-------------------Associate
       ---------------------------------[/comment]
  [if (e.oclIsKindOf(Association))]
  [let aAssociation : Association = e.oclAsType(
       Association)]
  [comment]-------------------order
       --------------------------------------[/comment]
  [if (aAssociation.name='order')]
  [let filter : String = genSelect('CustOrder.class')]
  [genDAttrFilter('orders', 'Type.Collection',filter, '
       false')/]
  [/let]
  [let associate : String = genAssociate('CustOrder.
       class', 0, 30, 'false')]
  [genDAssoc('create-order', 'mgmt', 'AssocTye.One2Many'
       , 'AssocEndType.One', associate)/]
  [/let]
  private Collection <[aAssociation.name.toUpperFirst()
       /]> [aAssociation.name/];
  [/if]
  [comment]----------------fill order
       --------------------------------------[/comment]
  [if (aAssociation.name='fillOrder')]
  [let filter : String = genSelect('FillOrder.class')]
  [genDAttrFilter('fillOrders', 'Type.Collection',filter
       , 'false')/]
  [/let]
  [let associate : String = genAssociate('FillOrder.
       class', 0, 30, 'false')]
  [genDAssoc('fill-order', 'mgmt', 'AssocTye.One2Many',
       'AssocEndType.One', associate)/]
  [/let]
  private Collection <[aAssociation.name.toUpperFirst()
       /]> [aAssociation.name/];
  [/if]

  [/let]
  [/if]
  [/for]
}
[/template]

[comment]------------------------ Queries DCSL
     --------------- --------[/comment]
[query public genDClass(serialisable: String,singleton:
     String) :
Set(String)='@DClass(serialisable='.concat(serialisable)
     .concat(',singleton=').concat(singleton).concat(')'
     )
/]
[query public genDAttrID(name: String,idAttr: String,
     auto:String,type:String,length: Integer,optional:
     String,muatable:String) :
Set(String)='@DAttr(name="'.concat(name).concat('",id=')
     .concat(idAttr).concat(',auto=').concat(auto).
     concat(',type=').concat(type).concat(',length=').
     concat(length.toString()).concat(',optional=').
     concat(optional).concat(',mutable=').concat(
     muatable).concat(')')
/]
[query public genDAttrFilter(name: String,type: String,
     filter: String,serialisable: String) :
Set(String)='@DAttr(name="'.concat(name).concat('",type=
     ').concat(type).concat(',filter=').concat(filter).
     concat(',serialisable=').concat(serialisable).
     concat(')')
/]
[query public genSelect(clazz: String) :
Set(String)='@Select(clazz='.concat(clazz).concat(')')
/]
[query public genDAssoc(ascName: String,role: String,
     ascType:String,endType:String,associate: String) :
Set(String)='@DAttr(ascName="'.concat(ascName).concat('
     ",role="').concat(role).concat('",ascType=').concat
     (ascType).concat(',endType=').concat(endType).
     concat(',associate=').concat(associate).concat(')')
/]

[query public genAssociate(type: String, cardMin:
     Integer, CardMax: Integer, updateLink: String) :
Set(String)='@Associate(type='.concat(type).concat(',
     cardMin=').concat(cardMin.toString()).concat(',
     cardMax=').concat(CardMax.toString()).concat(',
     updateLink=').concat(updateLink).concat(')')
/]
```