On Transforming a High-Level Specification to an Executable Domain Model 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 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 support tool for our method using Acceleo to automatically transform 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. Our recent work, we incorporated domain behaviors into a domain model for a composition of both structural and behavioral aspects of the domain, namely AGL [3]. Realizing 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 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] 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 role as the input domain model to transforming 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 generated software through a case study to that shows 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 a 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 we provide a detailed discussion on our motivating example, specifying precisely domain models for a DDD approach and research questions. Section III briefly discusses how a specification at high-level abstraction and transformed into a DCSL and AGL specification. Section IV we present the transformation UML activity diagrams into AGL as we provide a detailed the algorithm: AD2AGL (input, output, computations) and the rules (templates in Acceleo to automate the generation process). Section V we demonstrate the tool support for transforming from UML Activity Diagram and UML/OCL class diagram specification into the AGL^+ specification and generating software from the AGL^+ specification. Section VI concludes the paper.

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."

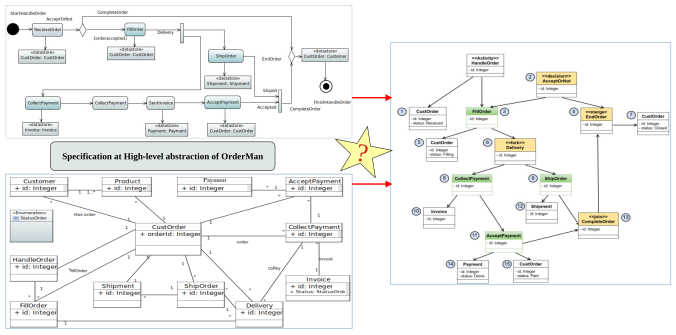


Fig. 1. Motivating Example

Our research problem, We deal with the domain problem using a four-step solution. Firstly, we specific a precise UML Activity diagram of the OrderMan and the guards for coordinate nodes. Secondly, we combined the UML Class diagram with the OCL constraints. Within our 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 show 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]. Then, Finally we execute AGL^+ of the domain model for DDD JDA: 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. 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 an HandleOrder (a new instance of the class `HandleOrder` is created) is received by the action `ReceiveOrder` (the operation `initHandleOrder`()) of the class `HandleOrder` is invoked). Then, if the guard condition (`Self.rejectOrder`()) of the decision node is not true (the order is not rejected), the flow goes 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. On the left path, the action ShipOrder (call of the operation `shipOrder`()) is executed. On the right path, to handle billing and payment processing, the action PayOrder (`Self.payOrder`()) is handled. When the two paths are accomplished, the join node may take place and the action EndOrder (`Self.EndOrder`()) is achieved. Returning back to the above decision node, if the order is rejected, the flow is passed directly to the action `EndOrder`.

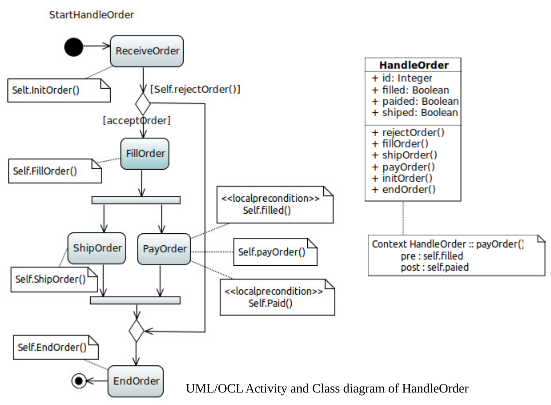


Fig. 2. The UML/OCL Activity and Class diagram of HandleOrder of OrderMan

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, the domain model tends to be the heart of software. Two main features of DDD are: (1) feasibility, i.e., a domain model should be the code and vice versa, and (2) satisfiability, i.e., the domain model would satisfy the domain requirements that are expressed in a ubiquitous language [1]. This language is defined for stakeholders, including the domain experts and developers, in an iterative and agile process of eliciting the domain requirements. 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 [14].

A DCSL and AGL specification

Annotation-Based Domain Specific Language (aDSL) is coined in [15] as an attempt to formalise the notion of fragmentary, internal DSL [16] 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 [17] and the C#. **Domain class specification language** (**DCSL**) [4] is a horizontal aDSL developed by us to express domain models. There are an *annotation-based domain specific language* (aDSL) [15] named *Domain class specification language* (DCSL) in order to express the domain models.

In this work we use feature of DCSL is that its meta-concepts model the domain-specific terms composed of the core OOPL meta-concepts and constraints. More specifically, the meta-concept **Domain Class** is composed of the meta-concept `Class` and a constraint captured by an annotation named `DClass`, which states whether or not the class is mutable. Similarly, the meta-concept **Domain Field** is composed of the meta-concept `Field` with a set of state space constraints, represented by an annotation named `DAttr`. The meta-concept `AssociativeField` represents the **Domain Field** that realizes one end of the association between two domain classes. DCSL supports all three types of association: one-to-one (abbr. one-one), one-to-many (abbr. one-many), and many-to-many (abbr. many-many). Finally, the meta-concept **Domain Method** is composed of `Method` and commonly used constraints and behavioral types that are imposed on instances of domain classes. The essential behavioral types are represented by an annotation named `DOpt` and another annotation named `AttrRef`. The latter references the domain field that is the primary subject of a method's behavior.

Out previous work [3], we proposed language, AGL (Activity Graph Language) to incorporating 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.

We create software artifacts following the DDD approach, we must combine DCSL and AGL specification namely AGL^+ specification, resulting in a Java program.

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

The activity diagram specified by the AGL for the `HandleOrder` Activity 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` is specified with `DClass.serialisable` = `false`. The domain field name is mutable `DAttr.mutable` = `false`.

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 proposes, 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 technique. This technique conceptually consists in iteratively performing three steps.

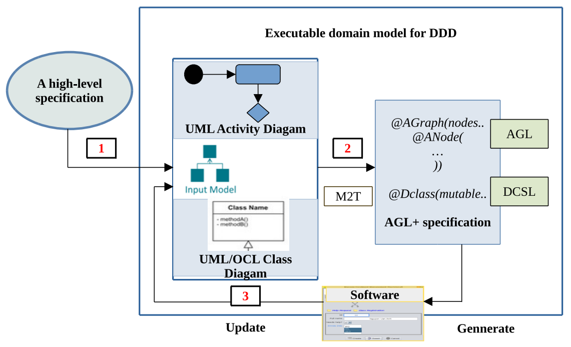


Fig. 3. Overview of Our Approach

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 to generate AGL^+ specification in source code from high-level specification AD and CD in Section IV. Here, we consider the AGL^+ specification as an extended domain model in MOSA [4]. It is expressed in AGL and DCSL as extends 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 updated 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 represent a algorithm to generate source code from high-level specification Activity diagram (AD) into the Activity Graph Language (AGL) so-call Algorithm AD2AGL. Algorithm .1 AD2AGL: input AD and output corresponding AGL, annotation realized in Java; e.g. `@AGraph`, `@ANodeand` `@MAct`, etc. First, it creates a java package with the name of the AD. Second, using algorithm Depth First Search on 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], activity diagram is considered as a graph. An activity graph [11], AG, is a hextuple which contains 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*, we refer the reader to Appendix A. Third, it checks each node in the AD and all action nodes and control nodes, and calls sub-functions for the respective cases to write the AGL specification. When visiting the current node, it calls the sub-functions *callsequentialAGL()*, *calldecisionAGL()*, *callforkAGL()*, *callmergeAGL()*, *calljoinAGL()* and *CallAction()* to visit the next node from the current node. we will illustrate the algorithm sub-algorithms for sequential node *callsequentialAGL()*, decision node *calldecisionAGL()*, fork node *callforkAGL()*, merge node *callmergeAGL()* and join node *calljoinAGL()*. The al-

Algorithm .1: AD2AGL- Transformation from Activity Diagram to Activity Graph Language

```
input  : AD a high-level specification
output : AGL specification
Create a Java package with the name of AD
  @AGraph(nodes={
Perform Depth First Search on AD keeping
  initialNode as the starting vertex.
      2.1 Visit node N, say $e_n$ ($e_n$ is the current
  node)
      2.2 Identify the the next node $e_n$ based on
  the edge, say $e_x$ ($e_x$ =callAction($e_n$),
  $x \in$ ($ControlNode$ :
  $d - decisionnode, f - fork\ node, m -$
  $mergenode, j - joinnode\ and\ s - sequentialnode$))
      2.3 Check the node $e_x$
           2.3.1 if ($e_x == e_s$)
                 callsequentialAGL()
                 Visit next node $e_x$
           2.3.2 if ($e_x == e_d$)
                 calldecisionAGL()
                 Visit next node $e_x$
           2.3.3. if ($e_x == e_f$)
                 callforkAGL()
                 Visit next node $e_x$
           2.3.4. if ($e_x == e_m$)
                 callmergeAGL()
                 Visit next node $e_x$
           2.3.5. if ($e_x == e_j$)
                 calljoinAGL()
                 Visit next node $e_x$
 Repeat step 2.1 to 2.3 until all nodes have been
  visited (meet finalNode)
  })
Return AGL
```

gorithm adds an AGL specification according to the annotation-based textual concrete syntax model for AGL that is defined in [3]. Four, it repeats step *callAction()* until all nodes have been visited (meet *finalNode*).

Sub-algorithm for sequential node

In the Algorithm .2 is the sub-algorithm for sequential node: *callsequentialAGL()*. This sub-algorithm takes the input sequential (a action node) specification in AD and depict a detailed of the nodes in Figure 4.

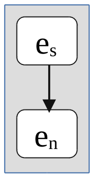
Sequential specification in Ad	**Desciption in algorithm: *callsequentialAGL()***
sequential pattern (n>1) 	the name of current node (action node): e_s
	the name of current node (action node): e_n
	the name of the incoming edge of the current node "as"

Fig. 4. The sequential specification in AD and denote in algorithm

Define sub-algorithm for sequential node

Algorithm .2: algorithm sub-algorithm for sequential node *callsequentialAGL()*

```
input  : Current node $e_s$
output : AGL sequential specification
1 2.3.1.1 Add a AGL specification code:
    ANode{refCls=name of node $e_s$,
    seviceCls=DataController, outClses=[the next
    node $e_s$, say $e_n$]
    actSeq=[MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls=name of node $e_n$,
    seviceCls=DataController, actSeq=[
    MAct{actName=newObject,
    pstStates=[NewObject]},
    MAct{actName=setDataFieldValue,
    fieldNames=["as"] pstStates=[Created]}]}
2 2.3.1.2    Visit next node $e_n$
3 2.3.1.3    Back to step 2.1
```

Sub-algorithm for decision node

In Algorithm .3, we present the sub-algorithm for the decision node, which we call *calldecisionAGL()*. This sub-algorithm takes the input decision specification in AD and depict a detailed of the nodes in Figure 5.

Sub-algorithm for fork node

In the Algorithm .4 is the algorithm sub-algorithm for fork node: *callforkAGL()* and it depicted in Figure 6.

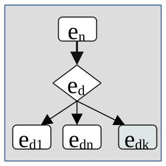
Decision specification in Ad	**Desciption in algorithm: *calldecisionAGL()***
decision pattern (if e_{dk} is present then *n>1* else *n>2*) 	the name of the current node (action node): e_n
	the name of the decision node: e_d
	the name of the nodes thats decision node link to: e_{d1}, e_{dn} and e_{dk} (e_{dk} is the data store node)

Fig. 5. The decision specification in AD and denote in Algorithm .3

Algorithm .3: Sub-algorithm for decision node *calldecisionAGL()*

```
input  : Current node $e_n$
output : AGL decision specification
1 2.3.2.1 Add a AGL specification code:
    @ANode{refCls=name of current node $e_n$,
    seviceCls=DataController, outClses=[the next
    node $e_n$, say $e_d$]
    actSeq=[@MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    @ANode{refCls=name of node $e_d$,
    nodeType=Decision seviceCls=DataController,
    outClses=[the next node $e_d$, say $e_{d1}$, $e_{dn}$ and data
    store node $e_{dk}$]}
    @ANode{refCls= name of node $e_{d1}$,
    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]}]}
    @ANode{refCls= $e_{dk}$,nodeType=$k$- specifies the
    control node type, outClses=$k_{out}$ - specifies the
    array of output $e_{dk}$}
2 2.3.2.2    Visit next node $e_n$
3 2.3.2.3    Back to step 2.1
```

Sub-algorithm for join node

In the Algorithm .5 is the algorithm sub-algorithm for join node: *calljoinAGL()* and it depicted in Figure 7.

Sub-algorithm for merge node

In the Algorithm .6 is the algorithm sub-algorithm for merge node: *callmergeAGL()* and it depicted in Figure 6.

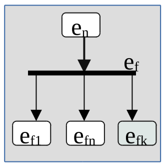
Fork specification in Ad	Desciption in algorithm: *callforkAGL()*
forked pattern (if e_{fk} is present then *n>1* else *n>2*) 	the name of the current node (action node): e_n
	the name of the fork node: e_f
	the name of the nodes thats fork node links to: e_{f1}, e_{fn} and e_{fk} (e_{fk} is the data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

Fig. 6. The fork specification in AD and denote in algorithm

Algorithm .4: algorithm sub-algorithm for fork node *callforkAGL()*

```
input  : Current node $e_n$
output : AGL fork specification
1 2.3.3.1 Add a AGL specification code:
    ANode{refCls=name of the current node $e_n$,
    seviceCls=DataController, outClses=[the next
    node $e_n$, say $e_f$]
    actSeq=[MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls=name of node $e_f$, nodeType=Fork
    seviceCls=DataController, outClses=[the next
    node $e_f$, say $e_{f1}$, $e_{fn}$ and data store node $e_{fk}$]}
    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]}]}
    ANode{refCls= $e_{fk}$,nodeType=$k$- specifies the
    control node type, outClses=$k_{out}$ - specifies the
    array of output $e_{dk}$ }
2 2.3.3.2    Visit next node $e_n$
3 2.3.3.3    Back to step 2.1
```

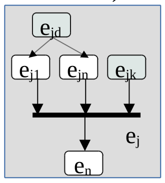
Join specification in Ad	Desciption in algorithm: *calljoinAGL()*
joined pattern (if e_{jk} is present then *n>1* else *n>2*) 	the name of the nodes thats join node links to: e_{j1}, e_{jn} and e_{jk} (e_{jk} is the data store node)
	- the name of the join node: e_j - the next node of e_j is e_n
	the name of the data store e_{jd} node, that link to: e_{j1}, e_{jn} (target of outgoing edge are e_{j1} and e_{jn})

Fig. 7. The join specification in AD and denote in algorithm

Algorithm .5: algorithm sub-algorithm join node *calljoinAGL()*

```
input  : Nodes $e_{j1}$, $e_{jn}$ and data store node $e_{jk}$
output : AGL join specification
1 2.3.4.1 Add a AGL specification code:
    ANode{refCls=name of the current node $e_{j1}$,
    seviceCls=DataController, outClses=[name of
    next node $e_{j1}$, say $e_j$]
    actSeq=[MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls=name of node $e_{jn}$,
    seviceCls=DataController, outClses=[name of the
    node $e_j$] actSeq=[ MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls= name of node $e_{jk}$, nodeType=$k$-
    specifies the control node type, outClses=[name of
    the node $e_j$], init=true}
    ANode{refCls=name of the node $e_j$,
    nodeType=Join, outClses=[name of a data store
    node of $e_j$, say $e_{jd}$]}
    ANode{refCls=name of the node $e_{jd}$,
    serviceCls=DataController actSeq=[
    MAct{actName=newObject,
    pstStates=[NewObject]},
    MAct{actName=setDataFieldValues,
    fieldNames=["a1","an"] pstStates=[Created]}] }
2 2.3.4.2    Visit next current node
3 2.3.4.3    Back to step 2.1
```

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. Rules transform 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]})
```

We define the rules (templates in Acceleo) for the transform of each kind of control node and action node in

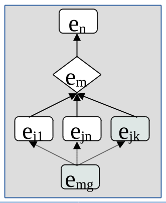
Merge specification in Ad	Desciption in algorithm: ***callmergeAGL()***
merged pattern (if e_{mk} is present then *n>1* else *n>2*) 	the name of the nodes thats join node links to: e_{m1}, e_{mn} and e_{mk} (e_{mk} is the data store node)
	- the name of the join node: e_m - the next node of e_m is e_n
	the name of the data store e_{mg} node, that link to: e_{j1}, e_{jn}, e_{mk} (target of outgoing edge are e_{j1}, e_{jn}, e_{mk})

Fig. 8. The merge specification in AD and denote in algorithm

Algorithm .6: algorithm sub-algorithm merge node *callmergeAGL()*

```
input  : Nodes $e_{m1}$, $e_{mn}$ and data store node $e_{mk}$
output : AGL merge specification
1 2.3.5.1 Add a AGL specification code:
    ANode{refCls=name of the current node $e_{m1}$,
    seviceCls=DataController, outClses=[The name of
    the node has a reference to the other node of the
    $e_m$, say $e_{mg}$]
    actSeq=[MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls=name of node $e_{mn}$,
    seviceCls=DataController, outClses=[name of the
    node $e_{mg}$] actSeq=[ MAct{actName=newObject,
    pstStates=[Created]}] init=true}
    ANode{refCls= name of node $e_{mk}$,
    nodeType=$k$- specifies the control node type,
    outClses=[name of the node $e_{mg}$], init=true}
    ANode{refCls=name of the node $e_{mg}$,
    nodeType=Merge, outClses=[name of merge node
    $e_m$]}
    ANode{refCls=name of the node $e_m$,
    serviceCls=DataController actSeq=[
    MAct{actName=newObject,
    pstStates=[NewObject]},
    MAct{actName=setDataFieldValues,
    fieldNames=["a1","an"] pstStates=[Created]}] }
2 2.3.5.2    Visit next current node
3 2.3.5.3    Back to step 2.1
```

the UML AD. We will illustrate the control node *Decision*, please refer in Appendix B the other nodes (*Sequential, Forked, Joined and Merged*).

Template transformation decision node in AD to AGL

The template in Acceleo to transform a *Decisional* node in AD, with sub-function *calldecisionAGL()*, is show in Listing 4.

Listing 4. Rules: transform decision node in AD to AGL

```
  /**visit next node*/
[template public callAction(currentNode : ActivityNode)]
  [currentNode.outgoing->any(true).target/]
[/template]
  /** Generate AGL specification for decision node*/
[template public callDecisionAGL(enNode : ActivityNode)]
[let edNode : ActivityNode = callAction(enNode)]
  @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]
```

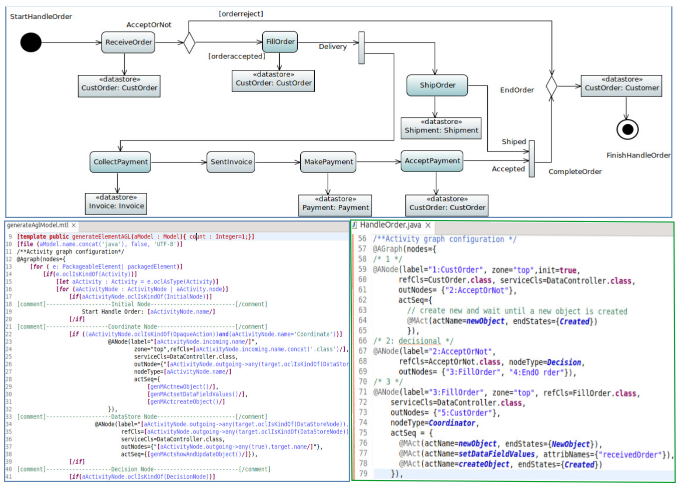
We use query templates to automatically generate **@MAct**. In Listing 5, a **@MAct** is generated code according to the query template.

Listing 5. Rules: generate the @*MAct*

```
  /**actName=newObject,pstStates={Created}*/
[query public genMActcreateObject() :
Set(String)='@MAct(actName=newObject,pstStates={Created})'/]
  /**Other @MActs*/
[query ...]
```

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 9 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.



```
[template public generateElementAGL(aModel : Model){ count : Integer=1;}]
[file (aModel.name.concat('java'), false, 'UTF-8')]
/**Activity graph configuration*/
@Agraph(nodes={
    [for ( e: PackageableElement| packagedElement)]
        [if(e.oclIsKindOf(Activity))]
            [let aActivity : Activity = e.oclAsType(Activity)]
            [for (aActivityNode : ActivityNode | aActivity.node)]
                [if(aActivityNode.oclIsKindOf(InitialNode))]
[comment]-------------------Initial Node------------------------[/comment]
                    Start Handle Order: [aActivityNode.name/]
                [/if]
[comment]------------------Coordinate Node----------------------[/comment]
                [if ((aActivityNode.oclIsKindOf(OpaqueAction))and(aActivityNode.name='Coordinate'))]
                        @ANode(label="[aActivityNode.incoming.name/]",
                            zone="top",refCls=[aActivityNode.incoming.name.concat('.class')/],
                            serviceCls=DataController.class,
                            outNode={"[aActivityNode.outgoing->any(target.oclIsKindOf(DataStor
                            nodeType=[aActivityNode.name/]
                            actSeq={
                                [genMActnewObject()/],
                                [genMActsetDataFieldValues()/],
                                [genMActcreateObject()/]
                        }),
[comment]-------------------DataStore Node----------------------[/comment]
                    @ANode(label="[aActivityNode.outgoing->any(target.oclIsKindOf(DataStoreNode)).
                        refCls=[aActivityNode.outgoing->any(target.oclIsKindOf(DataStoreNode))
                        serviceCls=DataController.class,
                        outNodes={"[aActivityNode.outgoing->any(true).target.name/]"},
                        actSeq={[genMActshowAndUpdateObject()/]}),
                [/if]
[comment]-------------------Decision Node-----------------------[/comment]
                [if(aActivityNode.oclIsKindOf(DecisionNode))]
```

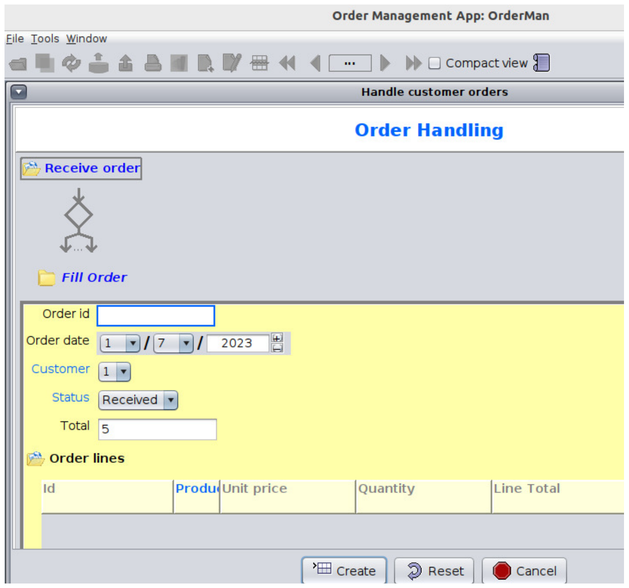
```
/**Activity graph configuration */
@AGraph(nodes={
/* 1 */
@ANode(label="1:CustOrder", zone="top",init=true,
      refCls=CustOrder.class, serviceCls=DataController.class,
      outNodes= {"2:AcceptOrNot"},
      actSeq={
        // create new and wait until a new object is created
        @MAct(actName=newObject, endStates={Created})
        }),
/* 2: decisional */
@ANode(label="2:AcceptOrNot",
      refCls=AcceptOrNot.class, nodeType=Decision,
      outNodes= {"3:FillOrder", "4:EndO rder"}),
/* 3 */
@ANode(label="3:FillOrder", zone="top", refCls=FillOrder.class,
    serviceCls=DataController.class,
    outNodes= {"5:CustOrder"},
    nodeType=Coordinator,
    actSeq = {
      @MAct(actName=newObject, endStates={NewObject}),
      @MAct(actName=setDataFieldValues, attribNames={"receivedOrder"}),
      @MAct(actName=createObject, endStates={Created})
    }),
```

Fig. 9. 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. We refer the reader to Appendix B of the technical report [18] accompanying this paper.

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 10 demonstrates the implementation of our method, using a supporting tool built on jDomainApp, a Java software framework.



Order id:
Order date: 1 / 7 / 2023
Customer: 1
Status: Received
Total: 5

Id	Produ	Unit price	Quantity	Line Total

Fig. 10. The GUI of the Orderman software generated by the tool.

VI. 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 20 (5) (2003) 19–25.
[3] 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.
[4] D. M. Le, D.-H. Dang, V.-H. Nguyen, On domain driven design using annotation-based domain specific language, Computer Languages, Systems & Structures 54 (2018) 199–235.
[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, S. Kent, Aspect-oriented metamodelling, The Computer Journal 46 (5) (2003) 566–577.
[8] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Synthesis lectures on software engineering 3 (1) (2017) 1–207.
[9] D. M. Le, D.-H. Dang, H. T. Vu, jdomainapp: a module-based domain-driven software framework, in: Proceedings of the 10th International Symposium on Information and Communication Technology, 2019, pp. 399–406.
[10] E. Sunitha, 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. doi:10.1109/ISDA.2012.6416605.
[11] S. E. Viswanathan, P. Samuel, Automatic code generation using unified modeling language activity and sequence models, Iet Software 10 (6) (2016) 164–172.
[12] Y. Rhazali, Y. Hadi, I. Chana, M. Lahmer, A. Rhattoy, A model transformation in model driven architecture from business model to web model, IAENG International Journal of Computer Science 45 (1) (2018) 104–117.
[13] V. Vernon, Implementing domain-driven design, Addison-Wesley, 2013.
[14] 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.
[15] M. Nosál, M. Sulír, J. Juhár, Language composition using source code annotations, Computer Science and Information Systems 13 (3) (2016) 707–729.
[16] M. Fowler, D.-S. Languages, Addison-wesley professional (2010).
[17] J. Gosling, The java language specification: Java se 8 edition, (No Title) (2015).
[18] V.-V. Le, N.-T. Be, D.-H. Dang, On Transforming a High-Level Specification to an Executable Domain Model for Domain-Driven Design, Tech. rep., VNU University of Engineering and Technology, Vietnam (2023).
URL `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 6. 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 7.

Listing 7. 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 8. Rules: transform fork node in AD to AGL

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

Template in Acceleo transformation merge node in AD to AGL

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

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

Template in Acceleo transformation join node in AD to AGL

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

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

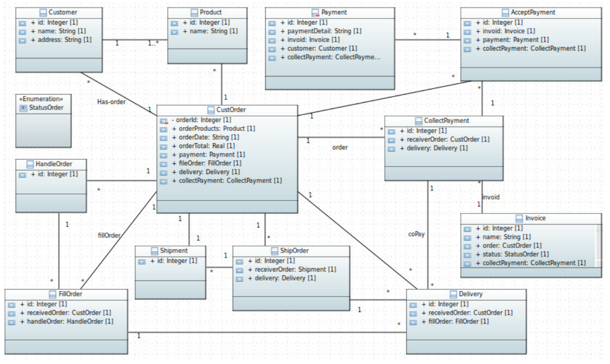


Fig. 11. UML/OCL Class diagram of OrderMan

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 11

```
/** 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(')')
/]
```