	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

## 1. ATL Transformation Example

### 1.1. Example: UML → Amble

The UML to Amble example describes a transformation from an UML model [1] specifying different aspects of a distributed program into an implementation of this distributed program in the Amble programming language [2].

Amble is a distributed programming language based on Objective Caml [3]. It enables to specify a distributed program in terms of processes that run concurrently. These processes are designed as states machines that are connected to each other by means of networks. Each network specifies a set of channels. The source process of a channel is allowed to write to its target process. An Amble program is composed of a single “.ml” file that includes the definition of the different process types that are involved in the program, as well as a set of “.topo” files (one for each considered network) that specifies the topology of the defined networks.

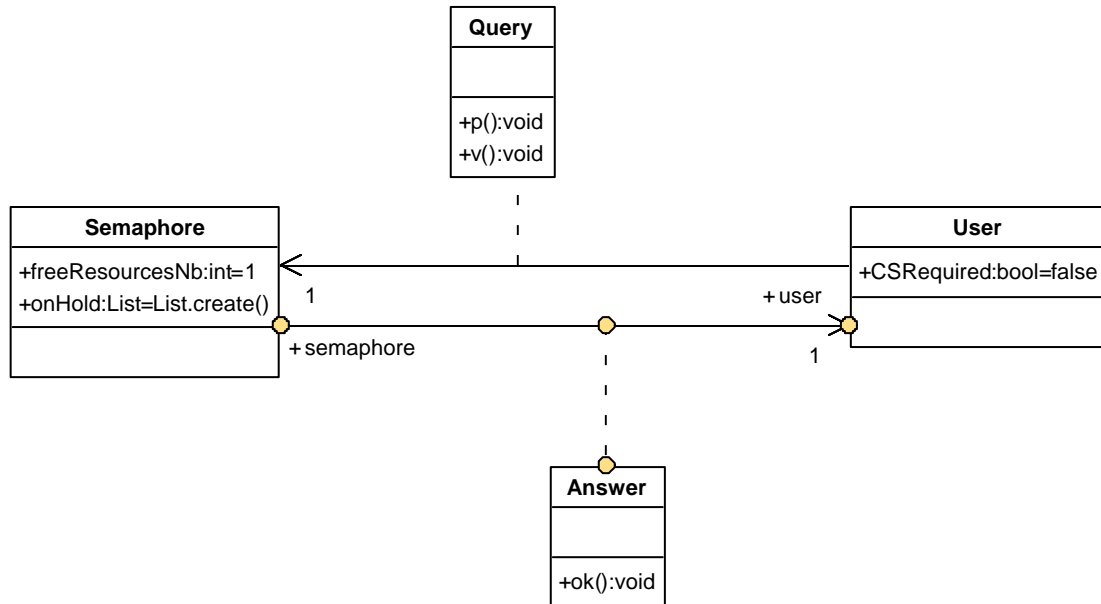
The input metamodel of this transformation is the UML metamodel. The transformation uses the definition of class and state diagrams. An Amble metamodel has been defined to be used as the output metamodel of the transformation. This metamodel describes the different elements that compose an Amble program and the way they are related.

An input UML model has to include a state diagram for each type of process involved in the Amble program to be generated, as well as a class diagram which describes the different kinds of considered processes (classes) and the existing communication channels (associations) between these channels.

The class diagram has to describe the different processes (one for each class) along with their local variables and methods. The classes can be connected to each other by means of unidirectional association classes that define the existing communication channels between the processes. Each association class defines a set of operations that correspond to the messages that can be sent over the channel. A states machine has to be provided for each defined class of the class diagram. The name of the states machines has to correspond to the one of the classes. Within a state machine, transitions are named “CONDITION”, “RECEIPT” or “RECEIPT\_ANY”, what corresponds to the different kinds of Amble transitions. The guard and the effect of a transition are encoded by the name of their respective UML model elements. Finally, when the guard of a transition corresponds to the receipt of a message, it has to be encoded by the name of the message prefixed by the string “received\_”.

### 1.2. An example input UML model

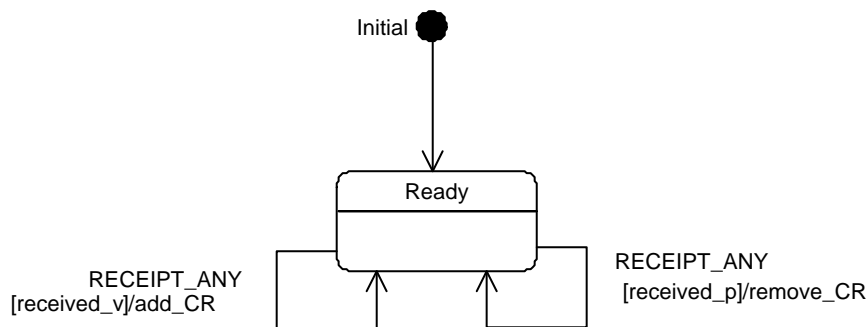
The example developed in this section corresponds to a semaphore-based synchronisation between several similar user processes that try to access a same critical resource. The input model is composed of two different kinds of processes: a Semaphore process and  $n$  User processes. The model is described in Figure 1, Figure 2 and Figure 3. The states machines provided in Figure 2 and Figure 3 respectively describe the behaviour of a Semaphore and a User process, whereas the class diagram provided in Figure 1 defines the processes local variables and the communication channels between the User and the Semaphore processes.



Created with Poseidon for UML Community Edition. Not for Commercial Use.

**Figure 1. Class diagram of the Semaphore example**


Figure 1 contains the class diagram associated with our example. The Semaphore process (here represented by a class) has two local variables encoding the number of available resources and the list of “on hold” client processes. A User process has, in this model, a single local variable that encodes whether it currently requires access to the critical resource. These two processes can communicate by means of two communication channels. The Query channel enables User processes to ask for and release a critical resource by means of the  $p()$  and  $v()$  messages whereas the Answer channel enables the Semaphore to notify a Process waiting for a critical resource that it is now available ( $ok()$  message).



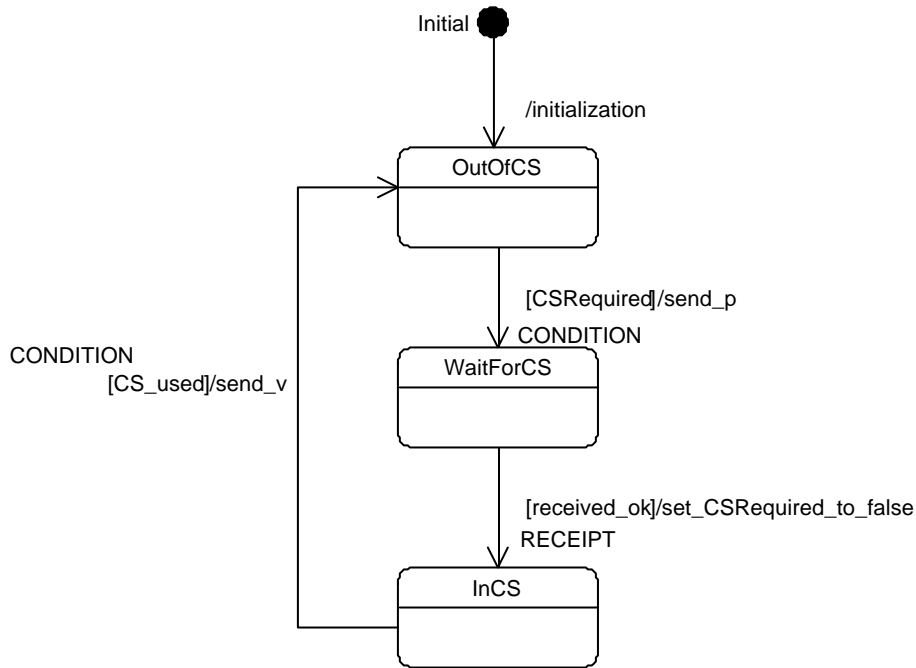
Created with Poseidon for UML Community Edition. Not for Commercial Use.

**Figure 2. State machine for the Semaphore process**

Figure 2 describes the behaviour of the Semaphore process. A Semaphore has a single state, Ready. It reacts to two kinds of events: the receipt of a  $p()$  or a  $v()$  message sent by any running processes (RECEIPT\_ANY). When a  $p()$  request is received, if there are available resources, the Semaphore sends an  $ok()$  message back to the client. Otherwise, this client is put into the *onHold* queue. When a  $v()$  message is received, a new resource is made available. If *onHold* is empty, *freeResourceNb* is

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

incremented. Otherwise, the first process of *onHold* is removed from the queue and an *ok()* message is sent to it.



Created with Poseidon for UML Community Edition. Not for Commercial Use.

**Figure 3. State machine for a User process**

Figure 3 describes the behaviour of a User process. A User process has three distinct states. The first one, *OutOfCS*, is the initial state. A process stays in this initial state as long as it does not require handling a critical resource. When it needs to access a critical resource (the condition *CSRequired* is fulfilled), it sends a *p()* request to the Semaphore process and steps to the *WaitForCS* state. The User process leaves this new state when is received an *ok()* message from the Semaphore process (RECEIPT, as opposed to RECEIPT\_ANY). It then steps to the *InCS* state and set its *CSRequired* variable to false. Once the critical resource has been used, it is released by sending a *v()* message to the Semaphore process. The process then returns to the *OutOfCS* state.

### 1.3. Metamodels

The UML to Amble transformation has the UML metamodel as input and the Amble metamodel as output. We describe here these two metamodels.

#### 1.3.1. The UML metamodel

In this example, we consider two simplified UML metamodels defining states machines and class diagrams. Both these metamodels are included in the UML metamodel which is not presented here. The transformation however makes use of the name property of the UML "Model" model element.

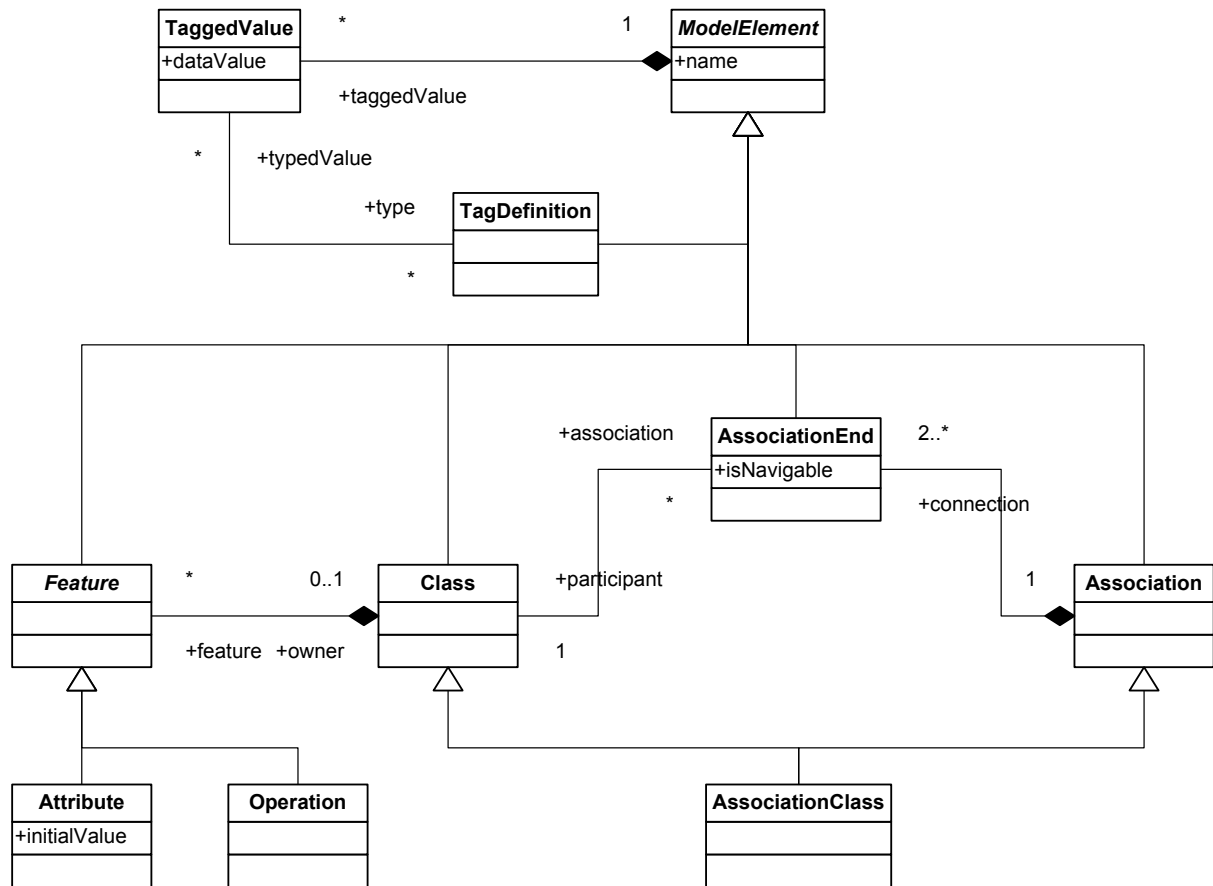

**Figure 4. Simplified UML class diagram metamodel**

Figure 4 describes the simplified UML class diagram metamodel used in the scope of this transformation. A Class can contain features that may be either Attributes (with an initialValue) or Operations (both inherit from the abstract Feature element). A Class may be associated with several Associations. An Association is connected to the Classes by means of AssociationEnd elements that may be navigable or not. The AssociationClass element inherits from both the Association and the Class elements.

Note that all elements, except TaggedValue, inherit from the abstract ModelElement element.

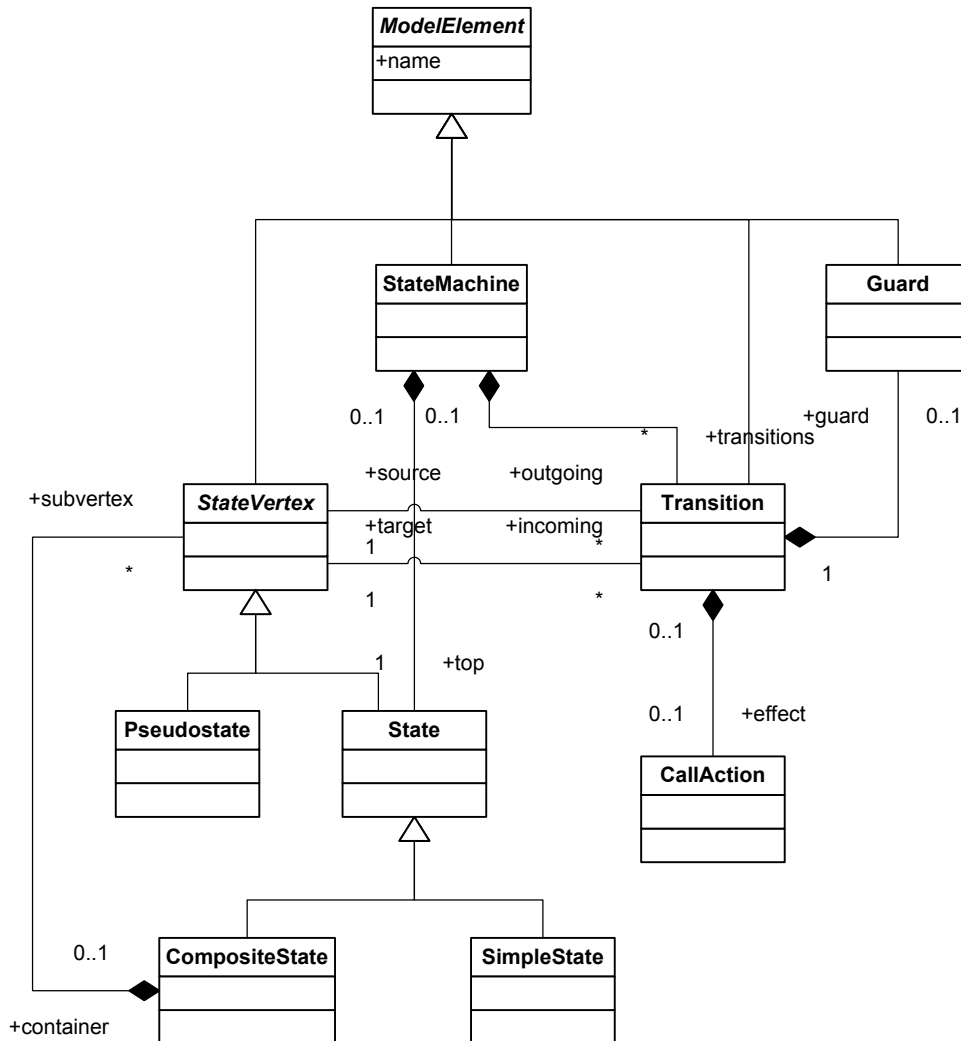

**Figure 5. Simplified UML state machine metamodel**

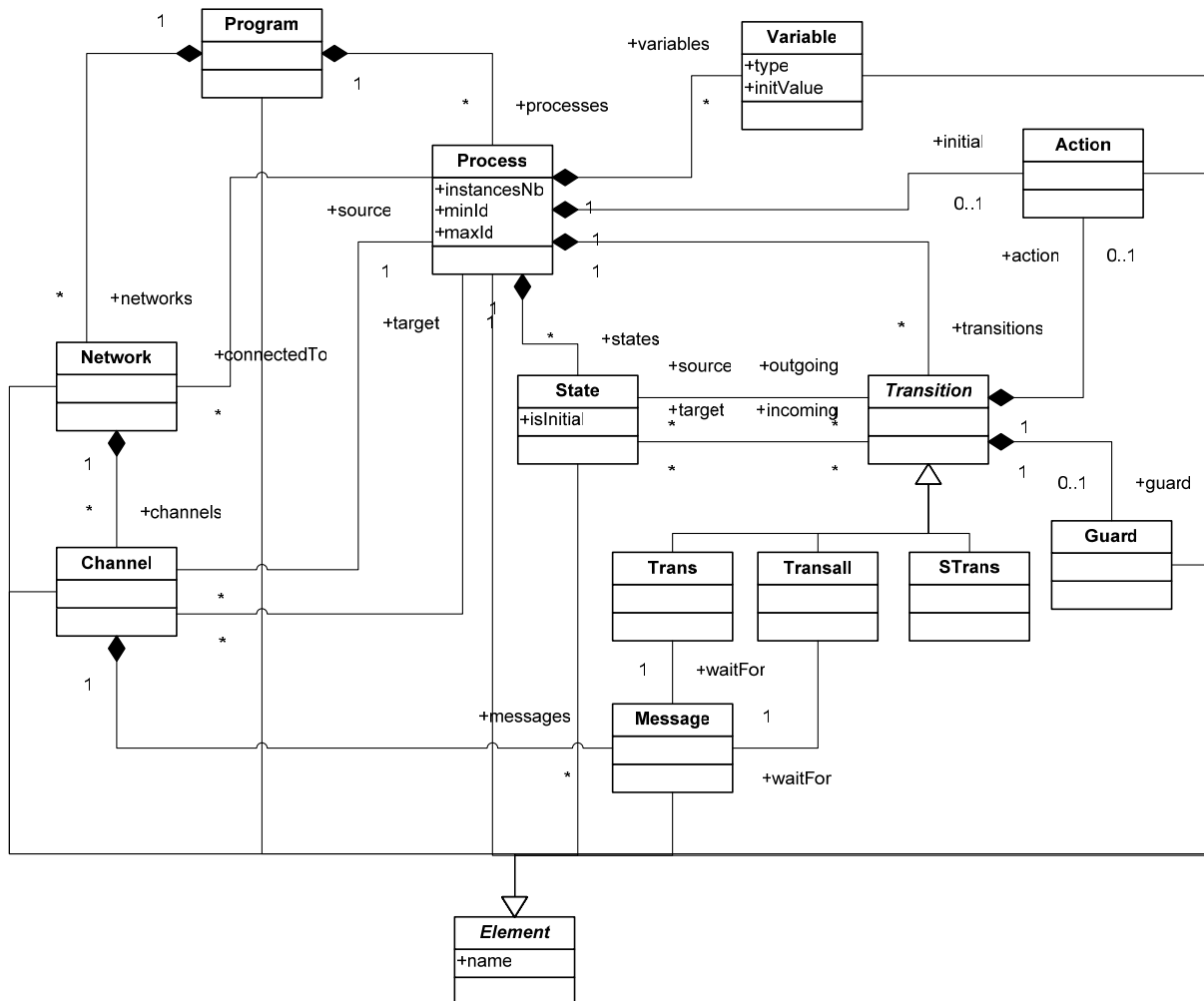
Figure 5 describes the simplified UML states machine metamodel used in the scope of the UML to Amble transformation. A StateMachine contains a one and only “top” State element and a set of Transition elements. The CompositeState and SimpleState elements both inherit from the State element. A CompositeState contains a set of abstract StateVertex elements. These abstract elements may be either State or PseudoState elements.

A Transition may contain a Guard and a CallAction. Note that CallAction is an UML 1.4 element that corresponds to the UML 1.5 Procedure element. A Transition has a single source StateVertex and a single target StateVertex. Each StateVertex can have several incoming and outgoing Transition elements.

Note that all elements, except CallAction, inherit from the abstract ModelElement element.

### 1.3.2. The Amble metamodel

The Amble metamodel describes the different model elements that compose an Amble model, as well as the way they can be linked to each other. The considered metamodel is presented in **Erreur ! Source du renvoi introuvable.** It is moreover provided in KM3 format [4] in Appendix I.



**Figure 6. The Amble metamodel**

A Program contains a set of Networks and a set of Processes. Each defined Process can be connected to several defined Networks. A Network defines a set of communication Channels. Each communication Channel has a source and a target Process. It is associated with a set of Messages that can be sent over the Channel from the source to the target Process.

A Program contains a set of States, a Set of Transitions, a set of Variables and an optional initial Action. The Variables define the local variables of the Process. They have a type and an initial value. The optional initial Action of the Process defines the action to be performed at initialization time. The set of States define the different states in which the Process may be during its execution. Each State may have several incoming and outgoing Transitions.

The set of Transitions contained by the Program describes the existing Transitions between the States of the Process. Each Transition has a single source State and target State. It may moreover contain an Action, which defines the action associated with the Transition, and a Guard that specifies a Boolean condition that has to be fulfilled for the Transition to be executed. Transition is an abstract model element which is of one of the following types:

- Trans defines a Transition that is triggered by the receipt of a Message sent by a given Process. It is associated with a particular type of Message.

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

- Transall defines a Transition that is triggered by the receipt of a Message sent by any Processes. It is associated with a particular type of Message.
- Strans defines a Transition that does not depend on the receipt of a Message.

Note that, except the Transitions elements, all the elements of this metamodel inherit from an abstract model element "Element" that defines the name of the element.


## 1.4. Rules Specification

Here are the main rules to transform an UML model into a distributed Amble program:

- For the UML **Model** element, an Amble Program element is created. It has the name of the Model and its processes and networks are respectively created for each StateMachine and AssociationClass.
- For each defined **StateMachine**, an Amble Process element is created. Its name and its transitions are copied from the input StateMachine element. Its number of instance is read from the TaggedValue attached to the Class homonym to the StateMachine. Its min and max Id are computed according to the rank of the Class in a Sequence of all involved Classes ordered by name. Thus, the minId of the first class of this sequence is 0, its maxId is the number of instances of the Class. minId of the second class then corresponds to the previous maxId + 1 and so on...

The set of States of the Process is computed by collecting all the SimpleState of the StateMachine. Its initialization Action is computed by finding the CallAction associated with the outgoing transition of the initial PseudoState of the StateMachine. The variables of the Process are computed by selecting all the Attributes that are owned by an UML Class element and whose owner's name is equal to the StateMachine name. Finally, the set of Networks the Process is connected to is computed by selecting all the AssociationClasses which point to the UML Class homonym to the StateMachine associated with the Process to be created.

- For each **Attribute** owned by a Class element, a Variable is created. Its name, type and initial value are copied from the input Attribute element.
- For each **SimpleState** element, an Amble State is created. Its name, as well as its links to the incoming and outgoing transitions, is copied from the input SimpleState element. The isInitial property is computed by checking if one of the states associated with the incoming transitions is a PseudoState (in UML, an initial state is represented by a PseudoState element).
- For each **Transition** which is named "RECEIPT", a Trans element is created. Its source, its target, its guard and its action are copied from the input Transition element. The UML Operation associated with the Message pointed by the Trans element is computed by selecting among the operations owned by an AssociationClass the one:
  - That has the same name (prefixed by "received\_") that the transition guard;
  - Whose owner's connections point to a Class that has the same name that the StateMachine the transition belongs to.
- For each **Transition** which is named "RECEIPT\_ANY", a Transall element is created. Its source, its target, its guard and its action are copied from the input Transition element. The Message associated with the Transall element is calculated in the same way that it is for a Trans element.

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

- For each **Transition** whose name is undefined or equal to "CONDITION", a STrans element is created. Its source, its target, its guard and its action are copied from the input Transition element.
- For each **Guard** element, an Amble Guard is created. Its name is copied from the input Guard element.
- For each **CallAction** element, an Effect is created. Its name is copied from the input CallAction element.
- For each **AssociationClass** element, a Network and a Channel are created. The Network name is copied from the AssociationClass name. It is associated with the Channel newly created. The Channel is named in the same way that the Network. The Messages that can be exchanged over the Channel correspond to the different features defined in the AssociationClass (it is supposed that an AssociationClass only contains Operations). The Channel source process corresponds to the StateMachine with the same name that the Class that is connected to the association at the non navigable end. The target process is computed in the same way with the navigable end.
- For each **Operation** owned by an AssociationClass element, a Message is created. Its name is copied from the input Operation element.

## 1.5. ATL Code

The ATL code for the UML to Amble transformation consists of 8 helpers and 11 rules.

### 1.5.1. Helpers

The first helper, **sortedClasses**, is a constant helper. It calculates a Sequence that contains all the Class model elements (but not the model elements derived from Class, such as AssociationClass elements) ordered according to their name. The computed sequence is used to determinate the min and max Id of each type of processes.


The **getInstancesNb()** helper computes the instances number defined for the contextual Class. This number is stored within the tagged value, whose tag definition name is "instances", associated with the Class element.

The **getMinId()** and **getMaxId()** helpers aim to compute the min and max Id of the process associated with the contextual Class. The multiple instances of a same process type are being assigned to consecutive Id values. In this scope, the **getMinId()** helper uses the **sortedClasses** sequence to compute its return value. This value corresponds to the sum of the instances number of the classes that appear before the contextual class in the **sortedClasses** sequence. The value returned by the **getMaxId()** helper is equal to the min Id of the class plus its instances number.

The **getVariables()** helper calculates the set of Attributes of a contextual states machine that correspond to the local variables of the process associated with the states machine. For this purpose, it simply selects all the Attribute instances that are owned by the Class model element (but not by model elements derived from Class) whose name is equal to the contextual states machine name.

When a Transition depends on the receipt of a message, the **getMessage()** helper aims to compute the Operation that defines to the received message within the UML class diagram. To this end, the helper selects, among all Operation instances whose owner is an AssociationClass model element, those:



	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

- Whose name, prefixed by the “received\_” string is equal to the name of the guard of the contextual transition.
- Whose AssociationClass is connected to a Class whose name is equals to the one of the state machine to which the contextual transition belongs to.

Finally, the **getSourceProcess()** and **getTargetProcess()** helpers respectively calculate the states machines that correspond to the source and target processes of the channel represented by the contextual AssociationClass. For this purpose, **getSourceProcess()** selects, among all states machines instances, the one that has the same name that the class connected to the navigable AssociationEnd of the contextual AssociationClass. **getTargetProcess()** behaves similarly, except that it selects the class connected to the non navigable AssociationEnd of the contextual AssociationClass.

### 1.5.2. Rules

Besides helpers, the UML to Amble transformation is composed of 11 rules.

The **Model2Program** rule generates an Amble Program element. Its name is copied from the input Model element, its processes and networks are generated for each StateMachine and AssociationClass input elements.

The **StateMachine2Process** rule aims to generate an Amble Process for each UML StateMachine. Name of the generated Process is copied from the state machine name. Its instancesNb, minId and maxId are respectively computed by the **getInstancesNb()**, **getMinId()** and **getMaxId()** helpers. The set of its local variables is computed by the **getVariables()** helper. Its set of states corresponds to the Amble States generated for each state of the input UML states machine. Its set of transitions corresponds to the transitions defined for the input UML states machine. Its initial action corresponds to the ActionCall associated with the outgoing transition of the UML initial PseudoState contained in the input states machine. Finally, the set of networks it is connected to contains the networks created for the UML AssociationClasses that are connected to an UML Class that has the same name that the input states machine.

The **Attribute2Variable** rule generates an Amble Variable for each UML Attribute owned by a Class model element.

The **State2State** rule generates an Amble State for each UML SimpleState. Its name, as well as its sets of incoming and outgoing transitions, is copied from the input SimpleState. Its isInitial property is computed by checking whether the input UML SimpleState is connected to the initial PseudoState of the stateq machine.

The **Transition2Trans** rule generates an Amble Trans element from each UML Transition whose name is “RECEIPT”. Source, target, guard and effect of the created Trans element are copied from the input UML Transition element. The message associated with this transition is computed by the **getMessage()** helper.

The **Transition2TransAll** rule generates an Amble Transall element from each UML Transition whose name is “RECEIPT\_ANY”. Source, target, guard and effect of the created Trans element are copied from the input UML Transition element. The message associated with this transition is computed by the **getMessage()** helper.

The **Transition2Strans** rule generates an Amble Strans element from each UML Transition whose source is an UML SimpleState element and whose name is either undefined or equal to “CONDITION”. Source, target, guard and effect of the created Trans element are copied from the input UML Transition element.

The **Guard2Guard** rule generates an Amble Guard for each UML Guard.

The **Effect2Action** rule generates an Amble Action for each UML CallAction.

The **Class2Network** rule generates both a Network and a Channel. The created elements all take the name of the input Class element. The created Channel constitutes the only channel of this network. The set of messages that can be sent over this channel corresponds to the features of the input AssociationClass. The source and the target of the channel are respectively computed by the **getSourceProcess()** and **getTargetProcess()** helpers.

The **Operation2Message** rule generates an Amble Message for each UML Operation owned by an AssociationClass model element.

```
1  module UML2Amble;
2  create OUT : Amble from IN : UMLDI;
3
4
5  -----
6  -- HELPERS -----
7  -----
8
9
10 -- This helper computes the sequence of Class model element ordered according
11 -- to their name.
12 -- CONTEXT: thisModule
13 -- RETURN: Sequence(UMLDI!Class)
14 helper def : sortedClasses : Sequence(UMLDI!Class) =
15     UMLDI!Class.allInstances()
16     ->select(a | a.oclIsTypeOf(UMLDI!Class))
17     ->asSequence()
18     ->sortedBy(b | b.name);
19
20
21 -- This helper calculates the number of declared instances of a given process.
22 -- This data is encoded as a taggedValue, of type 'instances' associated with
23 -- each Class model element.
24 -- CONTEXT: UMLDI!Class
25 -- RETURN: Integer
26 helper context UMLDI!Class def : getInstancesNb() : Integer =
27     self.taggedValue
28     ->select(x | x.type.name = 'instances')
29     ->first().dataValue
30     ->asSequence()->first().toInteger();
31
32
33 -- This helper calculates the minId of the process associated with the
34 -- contextual Class. Consecutive Ids are assigned to processes according to
35 -- their rank in the sortedClasses Sequence, and their instances number.
36 -- The minId associated with a Class C corresponds to the sum of instances
37 -- number of the classes that appear before C within sortedClasses.
38 -- CONTEXT: UMLDI!Class
39 -- RETURN: Integer
40 helper context UMLDI!Class def : getMinId() : Integer =
41     thisModule.sortedClasses
42     ->iterate(e; acc : Integer = 0 |
43         if thisModule.sortedClasses->indexOf(e)
44         < thisModule.sortedClasses->indexOf(self) then
45             acc + e.getInstancesNb()
46         else
47             acc
48         endif
49     );
```

```
50
51
52 -- This helper calculates the maxId of the process associated with the
53 -- contextual Class. This value corresponds to the minId of the Class
54 -- plus the instances number of the Class.
55 -- CONTEXT: UMLDI!Class
56 -- RETURN: Integer
57 helper context UMLDI!Class def : getMaxId() : Integer =
58     self.getMinId() + self.getInstancesNb() - 1;
59
60
61 -- This helper computes the set of attributes that are owned by the UML class
62 -- that has the same name that the contextual state machine.
63 -- CONTEXT: UMLDI!StateMachine
64 -- RETURN: Set(UMLDI!Attribute)
65 helper context UMLDI!StateMachine def : getVariables() : Set(UMLDI!Attribute) =
66     UMLDI!Attribute.allInstances()
67     ->select(a | a.owner.oclIsTypeOf(UMLDI!Class) and
68         a.owner.name = self.name
69     );
70
71
72 -- This helper computes the message receipt operation (owned by an association
73 -- class) associated with the contextual transition.
74 -- CONTEXT: UMLDI!Transition
75 -- RETURN: UMLDI!Operation
76 helper context UMLDI!Transition def : getMessage() : UMLDI!Operation =
77     let statemachine_name : String =
78         UMLDI!StateMachine.allInstances()
79         ->select(a | a.transitions->includes(self))
80         ->first().name in
81     let guard_name : String = self.guard.name in
82     UMLDI!Operation.allInstances()
83     ->select(a | a.owner.oclIsTypeOf(UMLDI!AssociationClass))
84     ->select(b | 'received_' + b.name = guard_name)
85     ->select(c | c.owner.connection
86         ->collect(d | d.participant)
87         ->collect(e | e.name)
88         ->includes(statemachine_name)
89     )
90     ->first();
91
92
93 -- This helper computes the state machine that has the same name that the
94 -- source class of the contextual association class.
95 -- CONTEXT: UMLDI!AssociationClass
96 -- RETURN: UMLDI!StateMachine
97 helper context UMLDI!AssociationClass
98     def : getSourceProcess() : UMLDI!StateMachine =
99     let source_name : String =
100         self.connection
101         ->select(a | not a.isNavigable)
102         ->first().participant.name in
103     UMLDI!StateMachine.allInstances()
104     ->select(a | a.name = source_name)->first();
105
106
107 -- This helper computes the state machine that has the same name that the
108 -- target class of the contextual association class.
109 -- CONTEXT: UMLDI!AssociationClass
110 -- RETURN: UMLDI!StateMachine
111 helper context UMLDI!AssociationClass
```

```
112     def : getTargetProcess() : UMLDI!StateMachine =
113     let target_name : String =
114         self.connection
115         ->select(a | a.isNavigable)
116         ->first().participant.name in
117     UMLDI!StateMachine.allInstances()
118     ->select(a | a.name = target_name)->first();
119
120
121
122 -----
123 -- RULES -----
124 -----
125
126
127 -- Rule 'Model2Program'.
128 -- This rule generates the structure of the root Program element from the UML
129 -- model.
130 rule Model2Program {
131     from
132     model: UMLDI!Model
133     to
134     prg: Amble!Program (
135         name <- model.name,
136         processes <- UMLDI!StateMachine.allInstances(),
137         networks <- UMLDI!AssociationClass.allInstances()
138     )
139 }
140
141
142 -- Rule 'StateMachine2Process'.
143 -- This rule generates an Amble process, with its states, its transitions and
144 -- its initial action from an UML state machine.
145 -- It also generates the 'id' variable associated with the created Amble
146 -- process.
147 rule StateMachine2Process {
148     from
149     statemachine: UMLDI!StateMachine
150     using {
151     crt_class : UMLDI!Class =
152         UMLDI!Class.allInstances()
153         ->select(a | a.name = statemachine.name)
154         ->first();
155     }
156     to
157     process: Amble!Process (
158         name <- statemachine.name,
159         instancesNb <- crt_class.getInstancesNb(),
160         minId <- crt_class.getMinId(),
161         maxId <- crt_class.getMaxId(),
162         states <- statemachine.top.subvertex
163         ->select(d | d.ocIsKindOf(UMLDI!SimpleState)),
164         transitions <- statemachine.transitions,
165         initial <- statemachine.top.subvertex
166         ->select(d | d.ocIsKindOf(UMLDI!Pseudostate))
167         ->collect(o | o.outgoing
168             ->collect(e | e.effect)).flatten()->first(),
169         variables <- statemachine.getVariables(),
170         connectedTo <- UMLDI!AssociationClass.allInstances()
171         ->select(e | e.connection
172             ->collect(d | d.participant.name)
173             ->includes(statemachine.name))
```

```
174     )
175 }
176
177
178 -- Rule 'Attribute2Variable'.
179 -- This rule generates an Amble process Variable from each UML attribute
180 -- that is defined within the context of an UML Class.
181 rule Attribute2Variable {
182   from
183     attribute: UMLDI!Attribute (
184       attribute.owner.oclIsTypeOf(UMLDI!Class)
185     )
186   to
187     variable: Amble!Variable (
188       name <- attribute.name,
189       type <- attribute.type.name,
190       initialValue <- attribute.initialValue.body
191     )
192 }
193
194
195 -- Rule 'State2State'.
196 -- This rule generates an Amble state with its name and its incoming and
197 -- outgoing transitions from an UML simple state.
198 -- The 'isInitial' property is computed by checking if one of the incoming
199 -- transitions is initialized by an UML pseudostate.
200 rule State2State {
201   from
202     uml_state: UMLDI!SimpleState
203   to
204     amble_state: Amble!State (
205       name <- uml_state.name,
206       isInitial <- not uml_state.incoming
207         ->collect(e | e.source)
208         ->select(d | d.oclIsKindOf(UMLDI!Pseudostate))
209         ->isEmpty(),
210       incoming <- uml_state.incoming,
211       outgoing <- uml_state.outgoing
212     )
213 }
214
215
216 -- Rule 'Transition2Trans'.
217 -- This rule generates the structure of the root Program element when the
218 -- input element contains no monitors.
219 rule Transition2Trans {
220   from
221     uml_trans: UMLDI!Transition(
222       uml_trans.name = 'RECEIPT'
223     )
224   to
225     trans: Amble!Trans (
226       source <- uml_trans.source,
227       target <- uml_trans.target,
228       guard <- uml_trans.guard,
229       action <- uml_trans.effect,
230       waitFor <- uml_trans.getMessage()
231     )
232 }
233
234
235 -- Rule 'Transition2TransAll'.
```

```
236 -- This rule generates the structure of the root Program element when the
237 -- input element contains no monitors.
238 rule Transition2TransAll {
239   from
240     uml_trans: UMLDI!Transition(
241       uml_trans.name = 'RECEIPT_ANY'
242     )
243   to
244     trans: Amble!Transall (
245       source <- uml_trans.source,
246       target <- uml_trans.target,
247       guard <- uml_trans.guard,
248       action <- uml_trans.effect,
249       waitFor <- uml_trans.getMessage()
250     )
251 }
252
253
254 -- Rule 'Transition2Strans'.
255 -- This rule generates the structure of the root Program element when the
256 -- input element contains no monitors.
257 rule Transition2Strans {
258   from
259     uml_trans: UMLDI!Transition(
260       (
261         uml_trans.name.oclIsUndefined()
262         or uml_trans.name = 'CONDITION'
263       )
264       and
265       uml_trans.source->oclIsKindOf(UMLDI!SimpleState)
266     )
267   to
268     trans: Amble!Strans (
269       source <- uml_trans.source,
270       target <- uml_trans.target,
271       guard <- uml_trans.guard,
272       action <- uml_trans.effect
273     )
274 }
275
276
277 -- Rule 'Guard2Guard'.
278 -- This rule generates an Amble guard from an UML guard.
279 rule Guard2Guard {
280   from
281     uml_guard: UMLDI!Guard
282   to
283     amble_guard: Amble!Guard (
284       name <- uml_guard.name
285     )
286 }
287
288
289 -- Rule 'Effect2Action'.
290 -- This rule generates an Amble action from an UML effect.
291 rule Effect2Action {
292   from
293     effect: UMLDI!CallAction
294   to
295     action: Amble!Action (
296       name <- effect.name
297     )

```

```
298 }
299
300
301 -- Rule 'Class2Network'.
302 -- This rule generates ...
303 rule Class2Network {
304   from
305     class: UMLDI!AssociationClass
306   to
307     net: Amble!Network (
308       name <- class.name,
309       channels <- new_channel
310     ),
311     new_channel: Amble!Channel (
312       name <- class.name,
313       messages <- class.feature,
314       source <- class.getSourceProcess(),
315       target <- class.getTargetProcess()
316     )
317 }
318
319
320 -- Rule 'Operation2Message'.
321 -- This rule generates ...
322 rule Operation2Message {
323   from
324     operation: UMLDI!Operation (
325       operation.owner.oclIsTypeOf(UMLDI!AssociationClass)
326     )
327   to
328     message: Amble!Message (
329       name <- operation.name
330     )
331 }
```

## I. Amble metamodel in KM3 format

```
package Program {

  abstract class LocatedElement {
    attribute location : String;
  }

  abstract class NamedElement extends LocatedElement {
    attribute name : String;
  }

  abstract class Structure extends NamedElement {
    reference variables[*] ordered container : VariableDeclaration oppositeOf
structure;
  }

  abstract class ProcContainerElement extends Structure {
    reference procedures[*] ordered container : Procedure oppositeOf "container";
  }

  class Program extends ProcContainerElement {
    reference monitors[*] ordered container : Monitor oppositeOf program;
  }

  class Monitor extends ProcContainerElement {
    reference program : Program oppositeOf monitors;
  }

  -- Procedures
  class Procedure extends Structure {
    reference "container" : ProcContainerElement oppositeOf procedures;
    reference parameters[*] ordered container : Parameter oppositeOf procedure;
    reference statements[*] ordered container : Statement;
  }

  class VariableDeclaration extends NamedElement {
    reference type : Type;
    reference initialValue[0-1] container : Expression;
    reference structure : Structure oppositeOf variables;
  }

  class Parameter extends VariableDeclaration {
    attribute direction : Direction;
    reference procedure : Procedure oppositeOf parameters;
  }

  enumeration Direction {
    literal in;
    literal out;
  }
  -- End Procedures

  -- Types
  class Type extends NamedElement {
  }
  -- End Types

  -- Expressions
  abstract class Expression extends LocatedElement {
  }
}
```





# ATL TRANSFORMATION EXAMPLE

Uml to Amble

Date 27/05/2005

```
class VariableExp extends Expression {
  reference declaration : VariableDeclaration;
}

-- PropertyCalls
abstract class PropertyCallExp extends Expression {
  reference source container : Expression;
  attribute name : String;
}

class OperatorCallExp extends PropertyCallExp {
  reference right container : Expression;
}

class AttributeCallExp extends PropertyCallExp {
}

class ProcedureCallExp extends PropertyCallExp {
  reference arguments[*] ordered container : Expression;
}
-- End PropertyCalls

-- Literals
abstract class LiteralExp extends Expression {
}

class BooleanExp extends LiteralExp {
  attribute symbol : Boolean;
}

class IntegerExp extends LiteralExp {
  attribute symbol : Integer;
}
-- End Literals
-- End Expressions


-- Statements
abstract class Statement extends LocatedElement {
}

class AssignmentStat extends Statement {
  reference target container : VariableExp;
  reference value container : Expression;
}

class ConditionalStat extends Statement {
  reference condition container : Expression;
  reference thenStats[1-]* container : Statement;
  reference elseStats[*] container : Statement;
}

class WhileStat extends Statement {
  reference condition container : Expression;
  reference doStats[1-]* container : Statement;
}

class ExpressionStat extends Statement {
  reference expression container : Expression;
}
-- End Statements
}
```

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>Uml to Amble</b>	Date 27/05/2005

---

## References

- [1] Unified Modeling Language (UML), version 1.5.  
<http://www.omg.org/technology/documents/formal/uml.htm>.
- [2] Amble library. Documentation and source code available at <http://home.gna.org/amble/>.
- [3] Objective Caml. Documentation and source code available at <http://caml.inria.fr/ocaml/>.
- [4] KM3: Kernel MetaMetaModel. Available at <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/atl/index.html>.