Code Generation for UML 2 Activity Diagrams

Towards a Comprehensive Model-Driven Development Approach

Dominik Gessenharter and Martin Rauscher

Institute of Software Engineering and Compiler Construction,
Ulm University, Ulm Germany
{Dominik.Gessenharter,Martin.Rauscher}@uni-ulm.de

Abstract. Modeling static structure and modeling behavior are often regarded as two distinct topics, however, in UML[1] they are not. They are even tightly coupled as can be seen e. g. by looking at attributes: That an attribute holds values at runtime is defined within the *Classes* language unit whereas the act of setting or getting a concrete value of an attribute is defined in the *Actions* language unit.

Tool support for modeling static structure is far more advanced than for modeling behavior. In particular, further model processing for activities like transformations or code generation is in a rudimentary stage.

In this paper, we present an approach for code generation for activities preceded by model transformations. Besides advancing model-driven development to properly include behavior, our contribution also enhances structural modeling by providing generation of code for accessing structural features based on the UML semantics of *Actions*.

Keywords: UML, Actions, Activities, Code Generation.

1 Introduction

> "Modeling is the designing of software applications before coding. Modeling is an essential part of large software projects, and helpful to medium and even small projects as well. (. . .) Models help us by letting us work at a higher level of abstraction. A model may do this by hiding or masking details, bringing out the big picture, or by focusing on different aspects of the prototype." [16]

This characterization of modeling is verbalized in the introduction to the probably most widely-used modeling language – OMG's (Object Management Group) UML, which currently is the de facto standard for modeling software systems. "Built upon fundamental OO concepts including class and operation, it is a natural fit for object-oriented languages and environments." [16] Object oriented programming bases on objects which inherently couple data (values of attributes) and methods for data manipulation.

[1] Unified Modeling Language [18], http://www.uml.org

R.B. France et al. (Eds.): ECMFA 2011, LNCS 6698, pp. 205–220, 2011.

UML supports structural modeling [18, §7] as well as behavioral modeling by defining *Actions* [18, §11] and *Behaviors*. Actions are basic concepts provided by UML whereas behaviors are user-defined. Since *Activity* [18, §12] is the only behavior directly containing actions [2], it is essential for behavioral modeling.

The relation of classes, actions and activities can be seen in Fig. 1 showing the semantic areas of UML as three distinct composite layers. Each layer depends on lower layers, but not on upper ones. The bottom layer is structural. Actions are the behavioral base for the higher-level behavioral formalisms of UML contained in the top layer. Clearly mapping actions to the structural foundation makes it easy to define the semantics of behavioral formalisms based on actions [4].

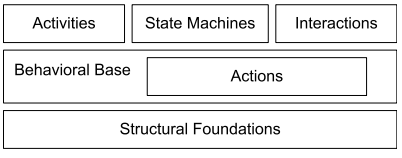


Fig. 1. A schematic of the UML semantic areas and their dependencies

Since modeling a system's structure and behavior gives rise to automatic generation of the full runnable code [11], UML is well suited for Model-Driven Development (MDD). In Sect. 2, we discuss modeling using UML at a glance. As for a comprehensive approach to MDD, tools are required supporting further processing of models [3] including both, structure and behavior, we present our code generation for static structures, *Actions*, and *Activities* in Sect. 3. To keep the code generation process straight forward, a preceding model transformation is used as in Sect. 4. We provide some results of an evaluation of our approach in Sect. 5, followed by a discussion of our contribution and related work.

2 Modeling with UML

2.1 Structural Modeling

The predominant elements of structural models are *Classes* and *Relationships* between classes, most often *Generalizations* and *Associations*. *Property* is used to define the structure of classes and associations. According to its upper bound, a property represents a single value or a collection of values when instantiated.

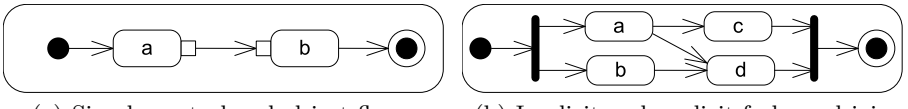
Another important feature available for classes only is *Operation*. It is a *BehavioralFeature* which may have a behavior associated [18, §13.2.22]. Calling an operation at runtime results in executing its specified behavior, e. g. an activity, a state machine or an *OpaqueBehavior*, i. e. a code fragment of any language.

2.2 Activities

UML defines the semantics of a number of specialized actions which serve as fundamental units of behavior specification. According to the UML metamodel [18]

and Bock [2], actions are directly contained only in activities. The sequence of action executions is defined by control flows or object flows which additionally provide input to actions from outputs of other actions (see Fig. 2(a)).

Our approach focuses on three concepts which make activities a very expressive formalism: *ObjectFlow*, concurrency, and *InterruptibleActivityRegions*. Dedicated object flows are convenient as they clearly show the locations of data creation and consumption and make activities well suited for modeling behaviors where object flows are extensively used. Concurrency may be a result of either explicit or implicit modeling. Fig. 2(b) shows the use of a *ForkNode* and a *JoinNode* for explicitly modeling concurrency and synchronization as well as an implicit fork (outgoing flows of `a`) and an implicit join (incoming flows of `d`). *InterruptibleActivityRegions* support aborting executions of actions which are grouped inside the region. Since aborting an execution might prevent locked resources from being released, aborting actions is risky.



(a) Simple control and object flows. (b) Implicit and explicit forks and joins

Fig. 2. Two examples of basic activities

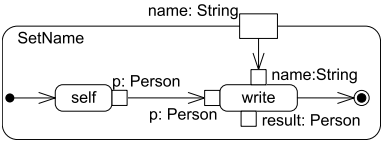
2.3 Actions

Actions are the fundamental units of executable functionality. UML's specification contains 37 concrete actions for various purposes. Specializations of *StructuralFeatureAction* support insertion and removal of single attribute values, removal of all values of an attribute and the retrieval of attribute values. Specializations of *LinkAction* provide similar functionality for links of associations. *CallOperationAction* causes a behavior which implements the operation to execute, *CallBehaviorAction* directly invokes another behavior.

Besides graphical modeling, UML encourages the use of a surface action language which encompasses both primitive actions and the control mechanisms provided by behaviors. Furthermore it may introduce higher-level constructs like e. g. a creation operation with initialization as a single unit as a shorthand for the create action to create an object and further actions to initialize attribute values and create objects for mandatory associations [18, §11, p.217].

Foundational UML (fUML) [19], a subset of UML, and the Action Language for Foundational UML (Alf) [17] both have been specified by the OMG. Although only very few new constructs have been introduced, using Alf may have advantages compared to traditional graphical modelling.

Fig. 3(a) shows an activity for setting the value of an attribute of the context object, Fig. 3(b) shows the same activity in Alf. Here, the graphical notation is more complex than a textual, code-like representation. The quicksort example of



(a) Graphical UML notation.

```
1 activity SetName(in _name: String)
2 {
3   this.name = _name;
4 }
```

(b) Implementation in Alf

Fig. 3. Graphical vs. textual representation of an activity writing an attribute value

the Alf specification [17, pp. 366, 368] contrasts an Alf implementation of 10 lines with a graphical representation consuming a whole page of this paper's format. Another advantage of Alf is that it can be seen which feature is updated. In the graphical notation of UML, this detail is not presented.

But even though the level of abstraction differs between graphical representation and Alf, two problems are still unsolved: 1) behavioral modeling in UML is based on very fine grained actions which are not suitable to reach a higher level of abstraction or bringing out a big picture and 2) the poor tool support of activities is not addressed by introducing another representation of activities.

2.4 Interplay of Structures, Actions and Behaviors

Behavioral models depend on structural models if actions access structural features. They may also serve as an implementation of operations contained in structural models. This is illustrated in Fig. 4: The metamodel is given on the left, a concrete instance on the right. Gray shaded beams connect meta classes with their instances in the model. The bold framed actions *read* and *write* are instances of specializations of the bold framed *StructuralFeatureAction*.

Dashed lines pointing from *StructuralFeatureActions* to associated *Structural Features* and from *Operations* to associated *Activities* represent instances of the bold printed meta associations between the corresponding meta classes.

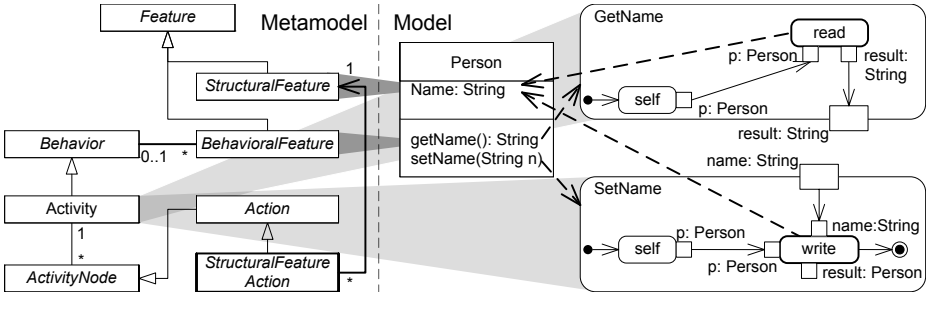


Fig. 4. Dependencies and relationships between metamodel and model elements

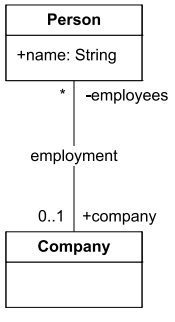
3 Code Generation for UML Models

3.1 Implementing Structural Models

Our code generator handles classes with attributes, associations and association classes based on the code patterns presented in our previous works [8]. For attributes, methods for writing and reading values are generated. If an attribute's upper bound is 1, `set` and `get` methods are generated, for upper bounds greater than 1, `add`, `get` and `remove` methods are generated. A `get` method returns a single value or a set of values according to the attribute's upper bound.

Associations are represented as dedicated classes managing a list of tuples representing the links between instances. Association classes are implemented by adding its features to the tuple representing the links.

Methods for creating or destroying links are contained in the implementation class of an association but are only visible to classes participating the association. Within these classes, `get` as well as `set` or `add` and `remove` methods are generated according to the association end's multiplicity, their visibility is determined by the end's visibility as shown in Fig. 5. The methods generated for association ends delegate calls to the association implementation class which creates or destroys tuples or arranges a result set of instances for `get` method calls.



(a) Class Diagram

```
1  public class employee{
2    private String name;
3    public void setName(String n){...}
4    public String getName(){...}
5    public void setCompany(Company c){...}
6    public Company getCompany(){...}
7  }
8  public class Company{
9    private void addEmployees(Employee e){...}
10   private void removeEmployees(Employee e){...}
11   private Employee[] getEmployees(){...}
12 }
```

(b) Implementation of classes Employee and Company

Fig. 5. An example of a structural model and its implementation. Note that for the association *employment* no implementation is contained in Fig. 5(b).

3.2 Code Generation for Actions

Generating code for accessing attributes or associations is close to an implementation of actions. To fit to the specification, `set`, `get`, `add`, and `remove` methods must be mapped to *StructuralFeatureActions* and *LinkActions* and thus be called when an *Action* associated with the corresponding attribute is executed.

Actions are implemented as part of the static structure by inserting code into those classes which are affected by an action, e. g. a *StructuralFeatureAction* is implemented in the class owning the associated feature.

In Sect. 2.3, we refer to the idea of using a surface action language. We avoid the development of such a language as well as supporting an existing one by using *OpaqueActions* for coupling models and code: for each *OpaqueAction*, an equally named method within the activity's context is called, i. e. code written by the user when implementing the method. Within this code, methods to which action executions are mapped can directly be called, i. e. actions can be used in models as well as in code. Generated code is located in generated classes whereas user written code is located in a subclass thereof. Thus, both kinds of code are separated from each other while generated code is still accessible to the user.

3.3 Basic Token Flow Concept

The semantics of the abstract metaclass *Action* from *FundamentalActivities* defines four steps of executing an action [18], of which the first is to create an action execution. The creation of an execution requires all object flow and control flow prerequisites to be satisfied, i. e. tokens must be available at all incoming edges. The second step is the consumption of the tokens which are removed from the original sources. The third step is executing the action until it terminates. After termination, the last step is to offer tokens to all outgoing edges.

A sequence of actions `a` and `b` with a single flow from `a` to `b` as shown in Fig. 6(a) is implemented as a sequence of statements (Fig. 6(b)).

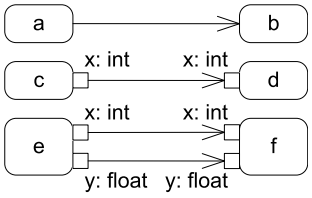
Object flows may be implemented by explicitly using a variable or by nesting calls, as shown for `c` and `d` in Fig. 6(b) and Fig. 6(c).

If multiple object flows exist between two actions, this still can be implemented as a sequence of statements. Depending on the implementation language, processing of the parameters requires appropriate techniques. Considering Java which only supports a single return value and no out parameters, a class is required to hold the return values as shown in Fig. 6(b). The class `XY` has two attributes representing the values of `x` and `y`, action `e` must create an instance of that class and write its output to the attributes. Fig. 6(c) shows an alternative implementation suitable for languages providing *out* parameters.

A basic flow, i. e. a flow from one action to another without any control nodes in between, sequences actions so that two sequential statements calling the according methods is a suitable implementation. Such sequences can be implemented in a single thread and as threads may be executed concurrently, a flow of multiple tokens can be implemented. For a proper implementation of the token flow semantics, it is necessary to properly implement guards as well as control nodes, as explained in the following.

3.4 Guards

If the guard of an edge does not hold, it prevents tokens from traversing that edge. Applied to Fig. 6, if a guard is annotated to the edge from `a` to `b` (or `c` to `d` or `e` to `f` respectively) the call of `b()` (`d()` or `f()`) is deferred as long as the guard is not satisfied. A proper implementation requires to pause the execution of code, if a guard prevents a token from flowing. This is important in particular



(a)

```
a(); b();
```

(b)

```
int x=c(); d(x);
```

```
XY xy = e();
f(xy.x, xy.y);
```

(b)

```
d(c());
```

```
int x; float y;
e(out x, out y); f(x,y);
```

(c)

Fig. 6. Three examples of sequences of actions and implementations

if concurrency occurs in an activity: The evaluation of a guard may depend on other concurrently executed sequences of actions. An implementation using `if` is insufficient as this does not cause the control flow of an application to pause.

Regarding Fig. 6, inserting `while (!<guard>){;}` after `a();`, `c();`, and `e();` pauses the activity execution. A more sophisticated approach is to let a thread wait until the guard holds. As a waiting thread cannot evaluate the guard, all waiting threads must be notified if a guard might evaluate to true. Therefore, all actions writing values to attributes or variables contain code to notify waiting threads which immediately re-evaluate guards. Depending on the result of this evaluation, threads continue executing or waiting for another notification.

Besides deferring the execution of an action in a sequence of actions, guards have a far more complex impact on the execution of actions if used in combination with control nodes. This is discussed in detail later.

3.5 Token Flow at Control Nodes

UML defines four kinds of control nodes: *DecisionNode*, *MergeNode*, *ForkNode*, and *JoinNode*. Before going into detail, we roughly give the implication of control nodes. A *DecisionNode* is used to choose between multiple alternatives from which only one may be taken. A *ForkNode* splits an incoming flow into multiple concurrent outgoing flows. Multiple flows are combined to one single flow by a *MergeNode*. A token arriving at one incoming edge results in a token flow on the outgoing edge. When combining multiple flows with a *JoinNode*, a token must be available at each incoming edge to emit a single token at the outgoing edge.

Tokens cannot rest at control nodes. A control token always flows from one action to another action, a data token may flow to a central buffer. An exception to this rule is the *ForkNode*, where tokens may be buffered. We discuss this in detail when looking closer at fork nodes later.

Note that two or more incoming edges of an action represent an implicit join, i. e. tokens must be offered to all incoming edges. This implicit join may be made explicit either by modeling so or by a model transformation. Therefore, we consider activities to be free of implicit joins. Analogously, two or more outgoing edges represent an implicit fork. For the same reason as for implicit joins, we consider all forks to be explicitly modeled.

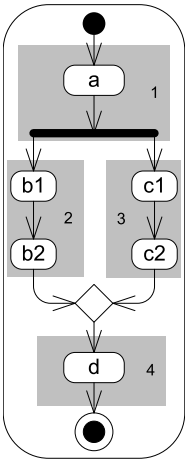
3.6 Code Generation Based on Token Flow Semantics

Generating code for sequences of actions is easy. When considering control nodes, the situation gets more complex, depending on how complex flows are composed: 1) at most one control node is used in a flow from one action to another; 2) flows may contain any quantity of control nodes, but the flow from the source to the target is acyclic; 3) any quantity of control nodes and cycles may occur.

The code generation as presented in this paper is designed for activities where flows contain at most one control node. However, it can be adapted to handle more complex flows, but remarkable effort is required. We provide additional information to this issue where necessary.

Our prototype is implemented in Java, but the general idea can be applied to any other object oriented language supporting threads. Sequences of actions are translated to sequences of method calls, each method being the implementation of the corresponding action (see above). Between sequences, control nodes occur so that at the end of each sequence, depending on the semantics of that control node, subsequent sequence(s) to execute must be determined and started.

As sequences might be executed concurrently, they must be implemented as dedicated threads. Separating the code for sequences in individual classes causes a large number of classes being created. We prefer to include all code in one class and to determine at runtime, which lines of code to execute. For this purpose, an `id` is introduced which is used to identify the sequence to run, as indicated by the grey boxes in Fig. 7. Setting the id to `-1` causes the thread to terminate.




```
1  class ActThread extends Thread {
2    private int id = 1;
3    public ActThread(int id) {
4      this.id = id;}
5    public void run() {
6      while(id >= 0) {
7        switch(id) {
8          case 1: a();
9                  new ActThread(2).start();
10                 id = 3;
11                 break;
12         case 2: b1();
13                 b2();
14                 id=4;
15                 break;
16         case 3: c1();
17                 c2();
18                 id=4;
19                 break;
20         case 4: d();
21                 id = -1;   } } } }
```

Fig. 7. An activity and the generated code for its implementation

Changing the id to another value causes the thread to execute another sequence of actions. If two sequences are to be started concurrently, a new thread is created with the appropriate id and the id of the current thread is changed according to the other sequence (cf Fig. 7, ll. 9–10). If a thread needs input data due to an object flow, `id` and data must be provided as parameters when creating it.

3.7 Token Flow at Control Nodes in Detail

Implementing a merge node is achieved by changing the value of `id`, as can be seen in Fig. 7, line 14 and line 18.

A token arriving at a decision node may traverse any of the outgoing edges, but only one of them. If guards are annotated to outgoing edges – what typically is the case as these guards define the decision to take – the token may traverse any edge the guard of which evaluates to true. If one of the outgoing edges is labeled with an `else` guard as is in Fig. 8(a), at least one edge may always be traversed. A suitable implementation is:

```
a();   if (x>0) id = ...;   else   id = ...;
```

Fig. 8(b) shows a situation in which none of the outgoing edges may be traversed if `x=0`. The decision shown in Fig. 8(c) is non-deterministic if `x=0`. To consider this while code generation requires an analysis of the guards which is difficult as guards may be any boolean expression evaluated within the context of the activity. We prefer to test guards of all edges for holding until a guard is found that is satisfied. If none of the guards holds, the thread waits until receiving a notification which is sent when attribute or variable values are changed.

```
while (true){
  if (<guard1>) { id = ...; break; }
  if (<guard2>) { id = ...; break; }
  ...
  wait(); }
```

A very complex semantics is that of the fork node. A token offered to its incoming edge is offered to the targets of its outgoing edges, if the corresponding guards hold. If at least one token is accepted by a target, a copy of this token is buffered at each outgoing edge which guard holds, but the target of which cannot accept the token. An example for this is given later on the basis of Fig. 9. If at most one control node appears between two actions, the implementation of the fork

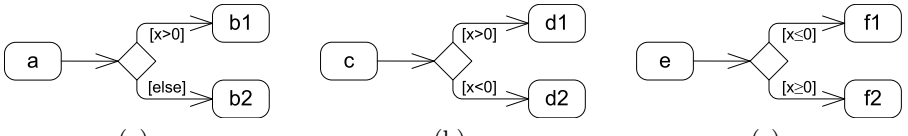
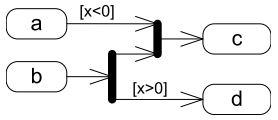


Fig. 8. Three examples of the use of a *DecisionNode*



(a)

```
while (!(<guard1>||<guard2>||...)){ wait(); }
if (<guard1>){ new ActThread(<id>).start(); }
if (<guard2>){ new ActThread(<id>).start(); }
...
```

(b)

Fig. 9. Use of fork and join nodes invalidating a simple fork node implementation

node is similar to that of the decision node as can be seen in Fig. 9(b). Items in angle brackets must be replaced by actual values.

If a flow from one action to another contains more than one control node, not only the guards of an edge determine whether a target action accepts a token. Subsequent *JoinNodes* may prevent a token from flowing, too. An example is given in Fig. 9(a).

A) If a token is offered only to the outgoing edge of `a` and `x<0` holds, `c` cannot accept that token as an additional token from `b` is required.

B) If a token is available at `b` but not at `a`, `d` is executed if `x>0` and a token is buffered at the edge from the fork node to the join node; if `x<0`, the offered token at `b` cannot flow.

C) If tokens are available from `a` and `b` and `x<0` holds, `c` is executed and no token is buffered at the fork node as the guard of the edge to `d` fails.

D) If, under conditions of C), `x>0` holds, `d` is executed and a token is buffered at the fork node. As soon as `x<0` holds, `c` is executed.

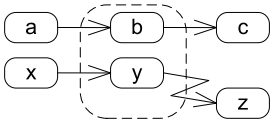
Note that the implementation of Fig. 9(b) is sufficient for a single fork node, but not if a fork node is used together with other control nodes in the same flow.

A proper implementation considering token buffering must represent control nodes as objects able to propagate tokens to subsequent control nodes and to indicate consumption of tokens to precedent nodes.

The *JoinNode* is the most complex node with regard to implementation. Although it is known which sequences must be executed before the join node is reached, it is not known, which threads will be the first ones reaching the join node. Our implementation of a join node is a class with lists of objects. Each list represents an incoming edge. Whenever one of the sequences ending at the join nodes terminates, a token is added to the appropriate list. After that, the join implementation checks all lists and if each list contains at least one token and all guards are satisfied, the next sequence is started and tokens are removed from the lists.

3.8 Implementing InterruptibleActivityRegions

When entering an *InterruptibleActivityRegion*, i. e. before executing the first action inside it, the current thread adds itself to a list of active threads inside the region (Fig. 10(b), l. 2). When leaving the region, the thread removes itself from the list as aborting the region no longer affects this thread (Fig. 10(b), l. 4). For convenience, the list is managed by an object representing the group (`ir1`).



(a)

```
1 case 1: a();
2   ir1.add(this);
3   b();
4   ir1.remove(this);
5   c();
6   break;
```

(b)

```
1 case 2: x();
2   ir1.add(this);
3   y();
4   ir1.remove(this);
5   ir1.abort();
6   z();
7   break;
```

(c)

Fig. 10. Code for supporting *InterruptibleActivityRegions*

The implementation of a sequence which leaves the region via an edge interrupting the region contains lines for adding and removing the thread in the list as well (Fig. 10(c), ll. 2 and 4). For actually causing all actions inside the region to be aborted, `ir1.abort();` is called (Fig. 10(c), l. 6). The implementation of that method might either kill all threads, sent a message requesting the threads to terminate, or set a flag which is checked before new actions are executed. One of the two latter options probably is preferable as killing threads – although closer to the specification – is quite risky.

A proper implementation of interruptible regions requires, when interrupted, to clear those lists of join nodes which represent edges having their source located inside the region and fork nodes within the region to discard buffered tokens.

4 Preparing Models by Model Transformations

A model transformation can keep our code generation straight forward by applying three steps: 1) make implicit forks and joins explicit; 2) move sequences such as shown in Fig. 6(a) to separate behaviors and replace them by *CallBehaviorActions*; 3) if possible, replace multiple control nodes by a single one.

An activity and the result of the transformation is shown in Fig. 11. Note that the two object flows between `a1` and `a2` are not handled as an implicit fork and subsequent join since both flows can be implemented as in Fig. 6.

The transformation A^t of activity A only contains *CallBehaviorActions* which implement simple sequences of actions. If A contained a *CallBehaviorAction* (say action `a2`), it was moved to activity X. Consequently, each *CallBehaviorAction* of A^t is mapped to a sequence of actions, each *CallBehaviorAction* occurring in a sequence represents a call of another behavior. All control nodes sequencing the sequences of actions remain in A^t. Thus, finding sequences is part of the transformation whereas sequencing of sequences is part of the code generation. Both conversions are solved purely programmatic by prototypically implementations.

Since all actions still persist – even though at a different location – mapping actions of A^t to actions of A can easily be done if necessary e. g. for the development of a debugger. In our approach, debugging is done by using our interpreter, but adding debug information to the generated code in principle is possible.

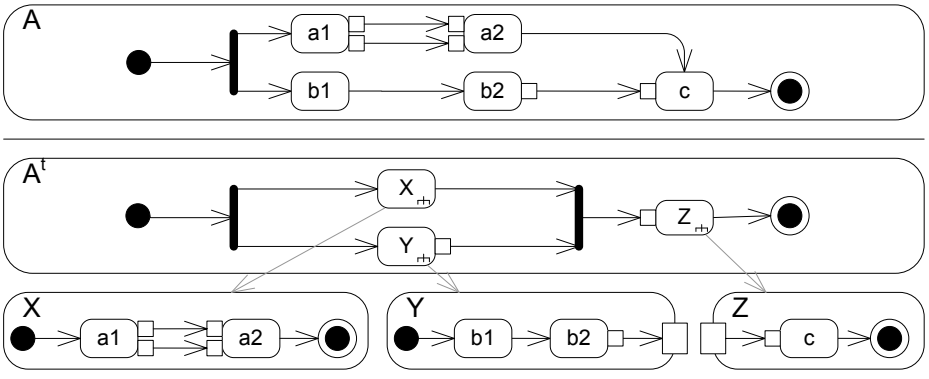


Fig. 11. Activity A and its transformation A^t with Activities X, Y, and Z

5 Evaluation

We consider the question of semantic correctness of model transformations and code generation as well as a comparison of generated code and model interpreters.

This does not include a formal verification of the semantic correctness, but our work is based on the formalization of the semantics of UML2 activities by Sarstedt [20] which we compared to the fUML specification. Since fUML is a subset of UML introducing additional constraints, its semantic is less complex. E. g. for fork nodes, the influence of guards on token buffering is dropped as fUML allows guards only at edges outgoing from a decision node.

The token flow formalization [20] serving as a foundation for our work is not UML compliant with regard to token buffering at fork nodes. This issue is considered in this work and tests show that for all elements presented here, the effects of generated code are equal to those of interpreters resulting from our own previous work as well as to that of a reference implementation of fUML[2].

An evaluation of our model transformation is done on a metalevel by the following consideration: moving sequences of actions to activities which are associated to a *CallBehaviorAction* is equivalent to the mapping of fUML where sequences of actions are moved into a structured node [19, §A.3.1].

Compared to Alf, UML activities are more expressive. E. g. in Alf, parallel flows are always arranged in a block structure, i. e. Fig. 2(b) cannot be expressed in Alf as in its syntax block statements may have a *parallel* annotation to force concurrent execution and synchronization of these blocks. In UML, *JoinNodes* wait for tokens on each incoming edge for synchronization. These tokens not necessarily have to be a copy of the same source token built at a *ForkNode*.

For evaluation of our prototype, we modeled an information system for public transportation of an imaginary city and generated code from that model. The structural model consists of 17 classes, 14 associations and 3 association classes,

[2] http://portal.modeldriven.org/content/fuml-reference-implementation-download

the behavioral model consists of a single activity of 14 Actions, 8 control flows and 15 object flows. Applying our code generation approach, the output is 1700 lines of code for the structural model including the implementation of actions and another 700 lines of code for the implementation of the activity.

The implementation of the static structure and of the behavior makes up more than 75% of the entire project code excluding the GUI. If excluding behavior, generated code amounts to only about 15%. For more details about the sample project as well as its implementation, we refer to our previous work [9].

Another aspect worth evaluating are runtime characteristics and size of the generated code. Numbers of these characteristics are only interesting if compared to those of other approaches. We compare our code generation approach with different kinds of activity interpreters: 1) an interpreter computing each token flow step by propagating tokens along edges; 2) an enhanced interpreter using lookup tables for flow computation – both results of our previous work – and 3) the fUML reference implementation.

The code generated by our prototype executes faster than interpreters as expected since interpreting causes additional overhead whereas generated code may be optimized by the compiler. The numerous console outputs of the fUML reference implementation were considered by adding console output to our code generator. Increasing complexity of models has fewer impact on the execution time of generated code. However, our own interpreters allocate less memory than the generated code if activities are complex; in case of small activities, memory allocation of generated code may be better as that of interpreters[3]. For a more detailed comparison of generated code and interpreters, we refer to our previous work [10].

6 Discussion and Related Work

In UML, only actions provide access to structural features. Some tools, e. g. Fujaba [25] provide access to attributes or even to associations by additional lines of code [7], which is a proprietary add-on not based on the UML specification. Such extra code can be seen as an implementation of actions if being mapped to the actions they represent, thus becoming applicable for behavioral modeling. Since *Activities* are the only behavior containing *Actions*, code implementing actions must be called when executing corresponding actions in an activity.

Our eclipse based prototype uses *UML2* of the *Modeling Development Tools (MDT)* [5] and can be coupled with a compatible editor such as *Eclipse UML2 Tools* or Papyrus [26] in order to provide a user friendly modeling interface with our model processing facilities. A survey giving a detailed analysis of current tools [6] reveals, that support of activities w.r.t. modeling is well supported in very few tools, e. g. Papyrus [26]. Further processing of activities, in particular their execution by interpreters or code generation is only rudimentary supported, if at all. If generating code for actions and compiling activities to code as presented, for some essential modeling elements the execution semantics of activities can be fully preserved when building applications from UML models.

[3] fUML reference implementation is excluded from this comparison

Activities are often associated with business processes. Our approach is probably not feasible for implementing business workflows. There are workflow engines available to which such processes might be deployed. But activities can also be used to describe short workflows were failover features with process state preservation etc. is not needed. E. g. the workflow of purchasing a ticket required to use a public transportation system: using a workflow engine for the process of selecting a departure and a destination station, a departure or arrival time and selecting a connection fitting the given constraints is oversized. For such purposes, we consider generating code a good idea.

Some advantages of generating code are that we can profit from compiler technique benefits like code optimizations which would have to be re-implemented in an interpreter if runtime characteristics of compiled code are desired. Not building an interpreter but transforming activities to code and thus making the Java virtual machine to the model interpreter is our goal.

Our approach supports implementations for accessing structural features and maps it to corresponding actions, thus achieving the coupling of structures and behaviors. By additionally accurately considering the subtleties of the UML token flow semantics, our approach supports behavioral modeling according to the specification. However, as composing complex flows by combining multiple control nodes between actions makes code generation very difficult – if not impossible – our approach is limited to a single control node between two actions. This limitation may be dropped when implementing control nodes as dedicated classes with methods for token propagation and token consumption.

Although other approaches dealing with code generation for activities exist, these approaches most often focus on some special application in which code generation for activities is only a minor part. UML based Web Engineering (UWE) [12,13] is concerned with this issue, but even claiming being based on standards, UWE only uses the notation but as can be seen in examples, the semantics of implicit joins is not considered [13, pp. 171,176][14]. Blu Age [23] which is based on executable UML suffers the same restriction.

Another approach for comprehensive MDD is implemented in UJECTOR. Since the provided examples only consist of simple sequences of actions [22, pp. 30,34], we can not assess the value of code generation for activities of this tool.

Sulistyo and Prinz propose *recursive modeling* to obtain complete code [21], but the introduced code patterns neither support concurrency nor a delay of token flow due to not holding guards.

Bhattacharjee and Shyamasundar present an approach for *validated code generation for activity diagrams* [1]. Mapping activities to Esterel, a language supporting concurrency, this approach overcomes limitations of the so far mentioned works. However, even here some problems are not addressed, e. g. the fact that it can not be statically decided which threads will be joined when reaching a join node. A delay of a token flow due to not satisfied guard conditions as well as the token buffering semantics of fork nodes are not considered.

Executable UML [15] gives detailed advice how to build executable UML models, but without addressing details of compiling models to code. BridgePoint[24]

is a tool based on executable UML, but modeling UML2 activities is not supported. Instead, Object Action Language (OAL) is used, which is less expressive than Alf as concurrency is not supported on the same level. Deferring an activity execution due to non satisfied guards is not possible in OAL, too. Thus, central concepts which we address in this paper are not applicable there.

Summing up we can find the token flow semantics of activities being not sufficiently implemented in current approaches or tools using more restrictive formalisms such as subsets of the UML specification of activities or action languages. As explained, a decision node cannot be implemented only by using an `if-then-else` statement. If the guard of each outgoing edge is not satisfied, the decision is delayed until one guard holds. Suchlike effects of the token flow semantics are very different to what programmers are used to and possibly therefore often not considered – but dealing with them is the essence of this contribution.

7 Conclusion and Future Work

Since code for behavior amounts for over 50% of the complete code of our sample project, better tool support for behaviors is a promising perspective. Furthermore, according to the specifiaction, activities couple behavior and structure. Thus, for comprehensive MDD, supporting activities is indispensible. Runtime characteristics of generated code show that including activities in an MDD approach by code generation offers some advantages which interpreters do not.

Our approach is currently limited with regard to a complex composition of flows by using control nodes. Pushing this boundary by analyzing complex flows which can – if matching special patterns – be implemented without explicit token propagation is our next step. Depending on the limitations remaining after that, a decision of whether implementing flows explicitly or not will be taken. A major objection is, that generating code for token propagation may become close to generating an interpreter which computes each step at runtime.

Apart from that, supporting additional modeling concepts like e. g. *Streaming* of object flows are further steps to take. Even architectural aspects like deployment raising the complexity of communication between instances might be considered, thus integrating behavior, structure and other aspects of UML.

References

1. Bhattacharjee, A., Shyamasundar, R.: Validated Code Generation for Activity Diagrams. In: Chakraborty, G. (ed.) ICDCIT 2005. LNCS, vol. 3816, pp. 508–521. Springer, Heidelberg (2005)
2. Bock, C.: UML 2 Activity and Action Models Part 2: Actions. Journal of Object Technology 2(5), 41–56 (2003)
3. Broy, M.: Challenges in Automotive Software Engineering. Proceedings of the ICSE 2006, pp. 33–42 (2006)
4. Crane, M.L.: Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification. PhD thesis, Queen's University, Kingston, Ontario, Canada (January 2009)

5. Eclipse Foundation, Inc. Eclipse Model Development Tools (MDT) (2011), http://www.eclipse.org
6. Eichelberger, H., Eldogan, Y., Schmid, K.: A Comprehensive Analysis of UML Tools, their Capabilities and their Compliance. Software Systems Engineering, Institut für Informatik, Universität Hildesheim (2009)
7. Fujaba Associations Specification (2005), http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/Fujaba/ Associations/~Specification
8. Gessenharter, D.: Implementing UML Associations in Java: A Slim Code Pattern for a Complex Modeling Concept. In: RAOOL 2009: Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages, pp. 17–24. ACM, New York (2009)
9. Gessenharter, D.: Extending The UML Semantics for a Better Support of Model Driven Software Development. In: Software Engineering Research and Practice, pp. 45–51 (2010)
10. Gessenharter, D.: UML Activities at Runtime - Experiences of Using Interpreters and Running Generated Code. In: Trujillo, J., Dobbie, G., Kangassalo, H., Hartmann, S., Kirchberg, M., Rossi, M., Reinhartz-Berger, I., Zimányi, E., Frasincar, F. (eds.) ER 2010. LNCS, vol. 6413, pp. 275–284. Springer, Heidelberg (2010)
11. Harel, D.: From Play-In Scenarios to Code: An Achievable Dream. Computer 34, 53–60 (2001)
12. Koch, N., Kraus, A.: The Expressive Power of UML-based Web Engineering (2002)
13. Koch, N., Zhang, G., Baumeister, H.: UML-Based Web Engineering: An Approach Based on Standards. In: Web Engineering: Modelling and Implementing Web Applications, pp. 157–191 (2008)
14. LMU: Ludwig-Maximilians-Universität München, Institute for Informatics Programming and Software Engineering. UWE Examples (December 2009), http://uwe.pst.ifi.lmu.de/exampleAddressBookWithContentUpdates.html
15. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston (2002); Foreword By-Jacoboson, Ivar
16. Object Management Group. Introduction to OMG's Unified Modeling Language (UML) (July 2005), http://www.omg.org/gettingstarted/what_is_uml.htm
17. Object Management Group. Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language, FTF Beta 1, OMG Document Number: ptc/2010-10-05 (2010)
18. Object Management Group. Unified Modeling Language (OMG UML), Superstructure Version 2.3 (2010), OMG Document Number: formal/2010-05-05
19. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0 (2011), OMG Document Number: formal/2011-02-01
20. Sarstedt, S.: Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams. PhD thesis, Ulm University (2006)
21. Sulistyo, S., Prinz, A.: Recursive Modeling for Completed Code Generation. In: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture, BM-MDA 2009, pp. 6:1–6:7. ACM, New York (2009)
22. Usman, M., Nadeem, A.: Automatic Generation of Java Code from UML Diagrams using UJECTOR. International Journal of Software Engineering and Its Applications 3(2), 21–37 (2009)
23. Blu Age Corp., Blu Age (2011), http://wiki.bluage.com/, §3.5.1
24. Mentor Graphics Corp., BridgePoint UML Suite (2010), http://www.mentor.com
25. Fujaba Development Group, Fujaba Tool Suite 4.3.2 (2007), http://www.fujaba.de/
26. Papyrus, Open Source Tool (2011), http://www.papyrusuml.org