An Execution Algorithm for UML Activity Graphs

Rik Eshuis* and Roel Wieringa

University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{eshuis,roelw}@cs.utwente.nl

Abstract. We present a real-time execution semantics for UML activity graphs that is intended for workflow modelling. The semantics is defined in terms of execution algorithms that define how components of a workflow system execute an activity graph. The semantics stays close to the semantics of UML state machines, but differs from it in some minor points. Our semantics deals with real time. The semantics provides a basis for verification of UML activity graphs, for example using model checking, and also for executing UML activity graphs using simulation tools. We illustrate an execution by means of a small example.

1 Introduction

A workflow model specifies an ordering on activities performed in an organisation. Typical ordering constructs are sequence, choice and parallelism. A useful notation for specifying this ordering is provided by UML activity graphs [14]. Figure 1 shows an example activity graph. Ovals represent activity states and rounded rectangles represent wait states. In an activity state, some activity is busy executing whereas in a wait state, an external event is waited for, e.g. a deadline must occur, or some third party must send some information. An activity state is called an action state in UML [14]. The workflow starts in the black dot (the initial state) and ends at the bull's eye (the end state). A bar represents a fork (more than one outgoing edge) or a join (more than one entering edge). A diamond represents a choice (more than one outgoing edge) or a merging of different choices (more than one entering edge).

The semantics in the current version 1.3 of UML is not yet entirably suitable for workflow modelling because in the UML 1.3 version, the semantics is specified in terms of state machines. (In UML 2.0, the semantics will be defined independently from state machines.) For example, in UML 1.3 an activity is defined as an entry action of a state. An entry action is executed to completion when its state is entered [14, p.2-144]. But in Fig. 1 this means that the two activities *Register* and *Evaluate* are executed simultaneously in the same run-to-completion step! This is not what we would like the activity graph of Fig. 1

* Supported by NWO/SION, grant nr. 612-62-02 (DAEMON).

M. Gogolla and C. Kobryn (Eds.): UML 2001, LNCS 2185, pp. 47–61, 2001.

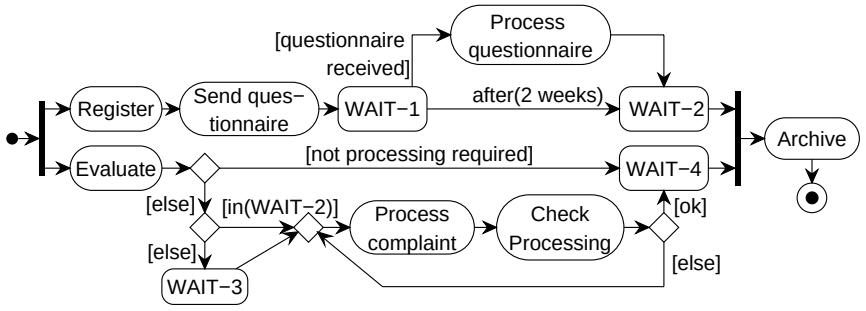

Fig. 1. Processing Complaints (adapted from Van der Aalst [1])

to say. What we would like to express by Fig. 1 is that *Register* and *Evaluate* start simultaneously, not that they should stop at the same time.

The underlying problem is that in UML 1.3, an activity graph is viewed as model of a software system that executes the activities. We want to use activity graphs for workflow modelling and therefore see an activity graph as a model of a workflow system (WFS). In workflow modelling, the activities are performed by actors (people or applications) external to the WFS, not by the WFS itself. It is the task of the WFS to monitor these activities, to manage the flow of data between them, and to route work items through a collection of actors, but it is not the task of the WFS to *execute* the activities. So in Fig. 1 the WFS executes the state transitions, i.e. the arrows in the diagram. The activities (nodes in the diagram) are executed by actors external to the WFS.

In this paper we present a formal execution semantics of UML activity graphs intended for workflow modelling that incorporates the above ideas. To show that our semantics is executable, we give our semantics in terms of an execution algorithm. Constraints are written in OCL and simple set theory.

The remainder of the paper is structured as follows. In Section 2 we explain in more detail how a WFS works. In Section 3 we define the syntax of activity graphs we use in this paper. In Section 4 we present our semantics in terms of execution algorithms for the components of the WFS that we identified in Section 2 and give a small example. In Section 5 we discuss related work. We end with conclusions and future work.

2 Workflow Systems

Purpose. A workflow system manages the flow of a case through an organisation. A case is the handling of a specific customer request to provide a certain service, e.g. the handling of an insurance claim in order to accept and pay the claim or to decline the claim. In a *case*, a certain set of activities is done in a certain sequence. An *activity* is an amount of work that is uninterruptible and that is performed in a non-zero span of time by an actor. An actor is either a user or an application. In an activity, the actor updates *case attributes*, which are stored in

a database. The sequence of the activities of a certain class of cases is specified in a *workflow model*. The WFS uses the sequencing information in the workflow model to route the case after an activity has terminated or event has occurred.

Events can be generated by the user of the WFS, the application used by this user, or by the database. The WFS reacts to the events by routing the case. We distinguish four kinds of events.

- A typical kind of event in a workflow is a *termination event*, which denotes that a certain activity has terminated (it is not important who the actor was) and that therefore the next activity can be started. What this next activity is, is determined by the WFS based on the WFS model. The WFS then routes the case to the next activity. Termination events are not defined in UML 1.3.
- In a slightly confusing terminology, a *completion event* is defined in the UML as the event when a wait state is entered[1] [14, p.2-147].
- An *external event* is a discrete change of some condition in the environment. This change can be referred to by giving a name to the change itself or to the condition that changes:
  - A *named external event* is an event that is given an unique name [14, p.2-131].
  - A *value change event* is an event that represents that a boolean condition has become true [14, p.2-131]. For example, in Fig. 1 the condition [questionnaire received] denotes a value change event.
- A *temporal event* is a moment in time to which the system is expected to respond, i.e. some deadline [14, p.2-131]. For example, in Fig. 1 the label after(2 weeks) denotes a deadline, after which the WFS is supposed to react by skipping the Process questionnaire activity. Temporal events are generated by the WFS itself. We assume that the WFS has an internal clock that measures the time.

In general, the state of a case is distributed over several actors. Each distributed part has a local state. Multiple instances of the same local state can be active at the same time. The (global) state of a case is therefore a bag, rather than a set, of local states.

Architecture. Our semantics is motivated by the following architecture of workflow systems (Fig. 2) [8,11,15]. It also resembles the architecture of UML state machines [14, p.2-149]. A WFS consists of two components, an event manager and a router, that act in parallel. These two components communicate with each other by means of a queue. The event manager receives events and puts them in the queue. The event manager is also responsible for generating temporal events.

[1] In UML 1.3 a completion event occurs when all entry actions and do-activities in the current state have completed. In this paper, however, wait states have no entry actions. Do-activities in UML 1.3 are activities performed by the software system. However, the activity states in our activity graph represent activities done by actors, not by the WFS. So we do not use do-activities.

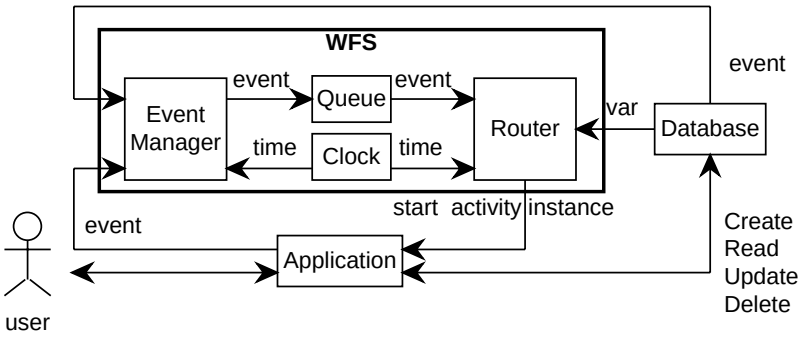

Fig. 2. Abstract execution architecture of a workflow system.

The router takes an event from the queue and routes the case. During routing typically new activities are started and new events are generated. It takes time for the router to process an event. Hence, when a new input event arrives, the router may be busy processing another input event. One of the outcomes of the routing may be that some new activities need to be started. This message is sent to the relevant actors. The assignment of activities to actors (resource management) falls outside the scope of this paper. The database typically generates change events. The user and/or the application typically generates termination and named external events. Only the router generates completion events.

3 Syntax

A UML activity graph consists of nodes and edges. Some of the nodes are only used to connect simple edges into more complex edges. Such nodes are called pseudo nodes and the complex edges they construct are called compound transitions in the UML [14, p.2-147]. The semantics is defined in terms of compound transitions. First we discuss the syntax of UML activity graphs. Next we discuss activity hypergraphs. An activity hypergraph is derived from a UML activity graph by replacing the pseudo state nodes by compound transitions.

UML activity graphs. In Section 1 we already explained the most important state nodes. In addition, a subactivity state node can be used to specify a compound activity. Each subactivity state node must have a corresponding activity graph that specifies the behaviour of the compound activity. In this paper we assume all subactivity state nodes have been eliminated from the activity graph by substituting for each subactivity state node its corresponding activity graph. The transitive closure of the hierarchy relation between activity graphs must therefore be acyclic.

Combining fork and merge, we can specify workflow models and patterns in which multiple instances of the same state node are active at the same time [2]. Figure 3 shows two example activity graphs in which a state node can have

multiple active instances. In the upper activity graph, B can be instantiated more than once whereas in the lower activity graph, C can be active twice at the same time if both A and B have terminated. State nodes (including pseudo state nodes) are linked by directed, labelled edges, expressing sequence. Each label has the form $e[g]/a$ where e is an event expression, g a guard expression and a an action expression. All three components are optional. An edge leaving an action state node cannot have an event label [14, p.2-164], since that would mean the atomic activity can be interrupted. The only action expression we allow is the sending of external events (broadcast). (Other action expressions would change the case attributes, which we do not want, since we want case attributes to be changed by actors, not by the WFS.) Special event labels when($time = texp$) and after($texp$) denote an absolute and a relative temporal event, respectively, where $texp$ is a natural number denoting time units.

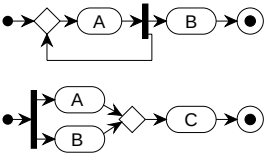

Fig. 3. Example activity graphs that can have multiple state instances

We do not treat object flow states since their semantics will be revised probably. Instead, we have local variables in our model that are stored in the database. We also have left out dynamic concurrency in our present definition, but it can be dealt with as indicated in our full report [6]. And we have left out swimlanes since these do not seem to impact the execution semantics.

Activity hypergraphs. To define a semantics for activity graphs, we must first flatten the activity graph into an activity hypergraph. Figure 4 shows the result of flattening Fig. 1 into an activity hypergraph. Figure 5 shows a meta model of activity hypergraphs. An activity hypergraph is a rooted directed hypergraph, consisting of nodes and hyperedges. A hyperedge is an edge that can have more than one source and more than one target.

A node is either an action state node, a wait state node, an initial state node, or a final state node. Every action state node has an associated activity. We use the convention that in the activity graph, an action state node is labelled with the name of its activity. If we use this name to indicate the node, we write

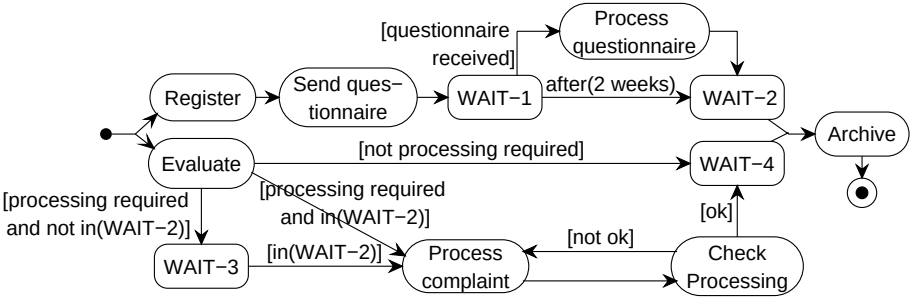

Fig. 4. Activity Hypergraph of Fig. 1

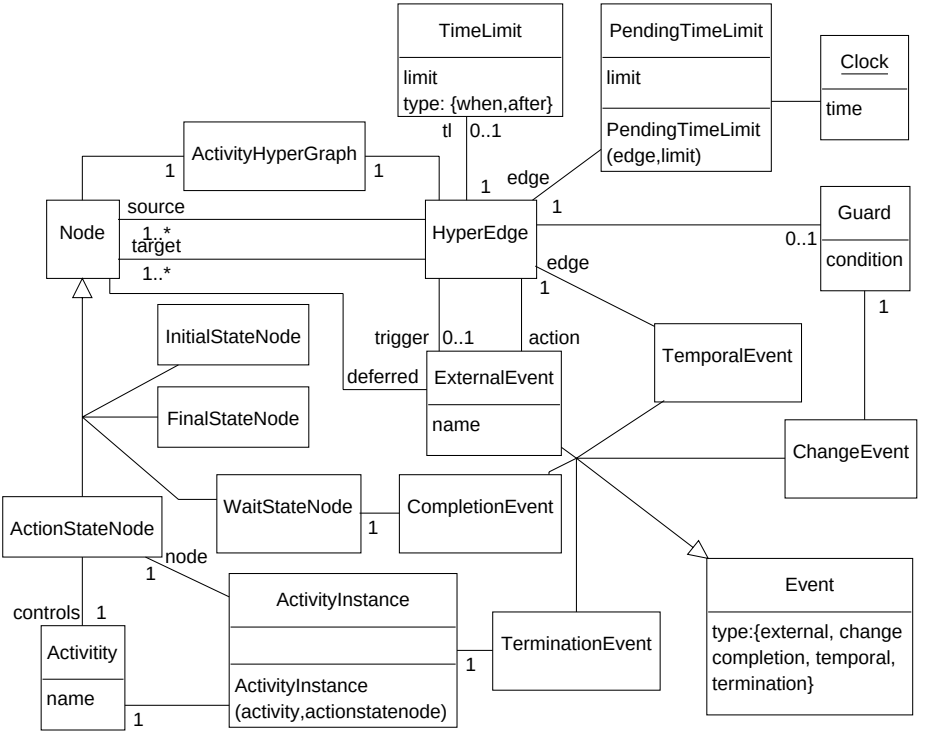

Fig. 5. Metamodel of activity hypergraphs

it in sans serif and if we use it to indicate the activity, we write it in *italic*. Different action state nodes may bear the same label, since they may have the same associated activity.

A hyperedge can have a trigger event, a guard and send actions. The trigger event must be a named external event. The only variables the guard can refer to are the case attributes of the workflow. A temporal event for a hyperedge is specified by declaring a time limit (class TimeLimit). At runtime a specification of a temporal event is translated into a deadline (class PendingTimeLimit) on the global clock (object Clock). If a hyperedge has a temporal event it must have no trigger event. Finally, termination events are implicitly specified by the fact that one of the source state nodes of an hyperedge is an action state node. In that case the action state node must have terminated before the hyperedge can be taken.

Mapping activity graphs to activity hypergraphs. An activity graph is mapped into a hypergraph by eliminating the pseudo state nodes of the activity graph using the concept of a compound transition [9,14]. Figure 6 shows some of the most common mappings. Roughly speaking, for every xor-node every pair of entering and exiting edges maps into one compound transition. In Fig. 4 for ex-

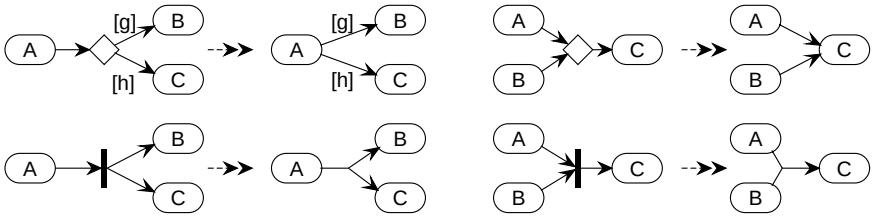

Fig. 6. Example eliminations of pseudo state nodes

ample, the decision node after Check processing has been replaced by two edges, one entering WAIT-4, the other one entering Process complaints. And for every and-node, all its entering and exiting edges map into the same compound transition. For example, in Fig. 4 the fork between the initial state node and Register and Evaluate has been replaced by one compound transition (or hyperedge), that has as source the initial state node, and as targets both Register and Evaluate. If xor-nodes are connected to and-nodes the mapping becomes slightly more complicated. The full report [6] gives all details.

4 Semantics

An execution of an activity hypergraph consists of a series of global states connected by transitions. A global state (or *configuration*) is a bag of currently active nodes. Remember that we need a bag of nodes rather than a set to allow multiple instances of a node to be active.

To define a transition we need the concept of relevance. A hyperedge is *relevant* in a configuration iff all its source state nodes are currently active, i.e. in the configuration, and all source action state nodes have terminated. For example, in Fig. 4 the hyperedge leaving Register is relevant iff the *Register* activity has terminated. Note that a wait state node does not have to be completed to make its leaving hyperedges relevant [14]. We call a node *finished* if it is either terminated (action state node) or completed (wait state node). The rules for taking a hyperedge are as follows.

– A hyperedge with either a trigger event or a temporal event, is taken when its trigger or temporal event occurs, it is relevant, and its guard is true.

For example, the hyperedge from WAIT-1 to WAIT-2 can only be taken if the temporal event after(2 weeks) has occurred.

– A hyperedge that has no trigger event but whose guard refers to some case attributes, is taken when
  - there occurs a change event that makes the guard true, and the hyperedge is relevant, or,
  - the last of the sources of the hyperedge have finished, it is relevant, and its guard is true by definition.

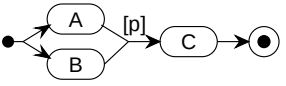

Fig. 7. Example enabling

For example, the hyperedge from WAIT-1 to Process questionnaire can only be taken if the change event that makes [questionnaire received] true has occurred. As a second, more complicated example, suppose in Fig. 7 the current configuration is [A,B] and A and B have not yet terminated. Now suppose A terminates but B does not. Then the bag of enabled hyperedges will still be empty, so the termination event of A will not trigger the hyperedge leaving {A,B}. Suppose next B terminates. Then the bag of enabled hyperedges will contain the hyperedge leaving {A,B} and entering C. But this hyperedge will only be taken if the guard p is true. If p is not true, the hyperedge can only be taken when next a change event occurs that makes p true.

– A hyperedge that has no trigger event and no guard, is taken when the last of its sources has finished and it is relevant.

If a hyperedge is taken, its source state nodes become inactive and its target state nodes become active.

We now give the details of our execution semantics for activity hypergraphs in terms of the abstract execution architecture depicted in Fig. 2. We present execution algorithms for both the event manager and the router. The components share the following variables:

– variable I represents the current bag of input events for the event manager,
– variable Q represents the event queue for the router,
– variable C represents the current configuration, which is a bag of state nodes,
– variable TL represents the set of temporal events that are scheduled to occur (pending time limits).

Event Manager. The event manager (Fig. 8) polls the current bag of input events in an infinite loop. If there are input events, these are put in the queue (operation + denotes bag union). Besides, the event manager generates time outs (lines 3-8) if the global clock has reached a certain value and adds these to the queue. Note that we do not use timers to generate time outs, since we

```
proc EventManager() ≡
1.  while true do
2.    if I ≠ bag {}  then Q:=Q+I; fi;
3.    foreach tl ∈ TL do
4.      if Clock.GetTime() ≥ tl  then
5.        var te:=new TemporalEvent(tl.edge);
6.        Q:=Q + bag { te } ;
7.      fi
8.    od
9.  od
```

Fig. 8. Procedure EventManager

```
while true do
1  Pick a non-deferred event from the queue
2  If it is a termination or completion event update F
3  Compute a step
4  Take a step:
   4.1  Update Q with the generated events
   4.2  Update TL by removing irrelevant scheduled time-outs and
        adding scheduled time-outs that have become relevant
   4.3  Create new activities
   4.4  Generate completion events
   4.5  Generate change events for the in predicate
   4.6  Update F and C
od
```

Fig. 9. Global structure of router

regard these to be implementation level constructs. Timers should not be used to specify temporal events but merely to help implementing them. It is possible that the event manager is slow and that a temporal event occurrence is added to Q after the global clock has reached its limit.

Router. In the router, we introduce an extra local variable F to store the bag of finished state nodes from C. Bag F is used to remember which state nodes from C have terminated or completed but could not yet been left (e.g. because some guard was false, or another source state node has not yet completed or terminated). For example in Fig. 7, when A terminates we add A to F. Note that keeping the termination event in the queue would mean the system gets stuck, since two termination events are needed before the hyperedge leaving {A,B} can be taken, and yet the router can handle only one (termination) event at a time. The global structure of the router is shown in Fig. 9. We next present each piece of the code, followed by an explanation of its meaning.

```
1  Pick a non-deferred event from the queue:
   proc Router() ≡
     1.  var F: bag Node;// the finished state nodes: C.includes(F)
     2.  F:=bag {};
     3.  while true do
     4.    if (Q≠ bag {}) then // there is an event to route
     5.      var event : Event;
     6.      event:=choose(highest(Q));// pick an event with the highest priority
     7.      if (!C → exists(n:Node|n.deferred → includes(event))) then
                                        // event is not deferred
     8.        var enabled,step : bag HyperEdge; // the bag of hyperedges that are
                                                  enabled and taken respectively
     9.        var newC : bag Node;// the new configuration
    10.        var newF : bag Node;// the new bag of finished state nodes
```

The router polls the queue in an infinite loop (lines 3-4). If the queue is not empty, the router picks the event with the highest priority (6). If there is more than one event that has highest priority, a nondeterministic choice between the candidate events is made. Note that usually some kind of fairness constraint must be imposed upon the queue to ensure that every event in the queue will eventually be processed. Here we have omitted such a constraint, since it is not present in UML 1.3. We assume operations *highest* and *choose* on finite bags. Operation *highest* returns the elements with the highest priority in the bag and operation *choose* returns nondeterministically an arbitrary element from a given finite bag. If the picked event is not deferred in the current configuration (7), it is processed (8-61). We use help variables $newC$ and $newF$ since below we refer to both the old and the new values of C and F.

2 If it is a termination or completion event update F:
11. $newF$:=F;
12. **if** (*event.type*=*termination*) **then**
13. $newF$:=$newF$ + bag {*event.activityinstance.node*}
14. **elsif** (*event.type*=*completion*) **then**
15. $newF$:=$newF$ + bag {*event.waitstatenode*}
16. **fi**;

Bag $newF$ is initialised with F. If the event is a termination or completion event, $newF$ is updated with the corresponding node.

3 Compute a step:
17. *enab*:=*ComputeEnabled(C,event,newF)*;
18. *step*:=a maximal, consistent subbag of *enab*;

The next step *step* is a maximal, consistent subbag of the bag *enab* of enabled hyperedges. We use procedure ComputeEnabled that we explain below in Fig. 10 to fill *enab*. Both the consistency and the maximality constraint are taken from the UML definition [14, p.2-151].

Roughly speaking, a bag of enabled hyperedges is inconsistent, given a configuration C if not all of them can be taken together, since the union of their sources is not contained in C. For example, in Fig. 1 if the current configuration contains Evaluate and the current event denotes termination of activity *Evaluate* then either the hyperedge to WAIT-4, or to Process Complaint or to WAIT-3 is taken, but none of these three hyperedges can be taken simultaneously, since then three instances of Evaluate would need to be active (and terminated!). So these three hyperedges are inconsistent.

Given a bag *enab* of enabled hyperedges, a consistent step *step* is maximal if there is no hyperedge $h \in enab$ but $h \notin step$ such that $step+bag\{h\}$ is consistent. So as many hyperedges as possible should be taken. In particular, if at least one hyperedge is enabled, the step cannot be empty. If more than one step exists, a random choice is made.

4.1 Update Q with the generated events:
19. **var** *generated*: set *Event*;// events generated when the step is taken
20. *generated*:= bag {};
21. **foreach** *e* ∈ *step* **do** *generated*:=*generated* + *e.action* **od**;
22. Q:=Q - bag {*event* } + *generated*;

Queue Q is updated with the bag of generated events. We use a help variable *generated* because Q is a shared variable that is also updated by the event manager. We assume that the assignment is atomic.

Next, we compute the value of help variables $newF$ and $newC$.

```
23.   newF:=newF - exited(C,step);
24.   newC:=nextconfig(C,step);
```

The new bag $newF$ of finished state nodes and the next configuration $newC$ are computed. Given a configuration C and a step S, $exited(C, S)$ denotes the bag of states that are exited when S is taken. Similarly, $entered(C, S)$ denotes the bag of states which are entered when S is taken. The next configuration when S is taken in configuration C will be $nextconfig(C, S) = C - exited(C, S) + entered(C, S)$.

4.2 Update *TL* by removing irrelevant scheduled time-outs and adding scheduled time-outs that have become relevant:

The code for this part is split in three.

```
25.   var OffTL: set PendingTimeLimit; // irrelevant timeouts
26.   var OldRelevant : bag HyperEdge;
27.   OldRelevant:=rel(C,F) - rel(newC,newF);
28.   foreach tl ∈ TL do
29.     if (tl.edge ∈ OldRelevant) then // tl has become irrelevant
30.       OffTL:=OffTL + bag { tl };
31.       OldRelevant:=OldRelevant - bag { tl.edge };
32.     fi
33.   od
```

First the set of irrelevant time outs is computed. A scheduled time out becomes irrelevant, if the hyperedge that the scheduled time out corresponds to is no longer relevant in the new configuration, i.e. the source of the hyperedge is no longer contained in the new relevant configuration. Consider for example the activity graph in Fig. 1. If the old configuration C contains WAIT-1 and the change event [questionnaire received] occurs, then the next configuration C' will contain Process questionnaire instead of WAIT-1. Hence, hyperedges leaving WAIT-1 are no longer relevant, and the scheduled time out after(2 weeks) that was relevant in C is irrelevant in C'. The *relevant* bag of hyperedges $rel(C, F)$ for a configuration C and a bag F of finished state nodes consists of those hyperedges whose non-action state node sources are contained in C and whose action state node sources have terminated, so are in F.

```
34.   var OnTL: set PendingTimeLimit; // new relevant timeouts
35.   var NewRelevant : bag HyperEdge;
36.   NewRelevant:=rel(newC,newF)-rel(C,F);
37.   foreach h ∈ NewRelevant do
38.     if (h.tl.notEmpty) then // there is a time out specified for h
39.       var ptl : PendingTimeLimit;
40.       if (h.tl.type=after) then // create a new pending time limit
                                    scheduled tl.limit time units from now
41.         ptl:=new PendingTimeLimit(e,h.tl.limit+Clock.time);
42.       else ptl:=new PendingTimeLimit(h,h.tl.limit);
```

```
43.              fi
44.            OnTL:=OnTL + bag { ptl };
45.            fi
46.          od
```

Next, the set $OnTL$ of new relevant scheduled time outs is computed. A time out becomes scheduled iff the hyperedge that the time out corresponds to was irrelevant in the old configuration but becomes relevant in the new configuration. For example, in Fig. 1, if the old configuration contains Send questionnaire and the activity *Send questionnaire* has terminated, then the new configuration contains WAIT-1 instead of Send questionnaire. Then time out after(2 weeks) becomes scheduled.

```
47.          TL:=TL-OffTL+OnTL;
48.          foreach t ∈ OffTL do delete t; od;
```

Finally, set TL is updated and the irrelevant scheduled time outs are deleted.

```
49.          foreach s ∈ entered(C,step) do
```

Each entered state node is checked in order to start new activities (4.3), generate completion events (4.4), and generate *in* change events (4.5).

```
4.3 Create new activities:
50.            if (s.oclIsTypeOf(ActionStateNode)) then // s is action state node
51.              var ais:=new ActivityInstance(s.controls,s) ;
52.              start(ais); // start this activity instance
```

For each action state node that is entered, a new activity instance is created and started.

```
4.4 Generate completion events:
53.            elsif (s.oclIsTypeOf(WaitStateNode)) then // s is wait state node
54.              generate completion event for s and add it to Q
55.            fi
```

If a wait state node is entered, its completion event is generated and inserted into Q.

```
4.5 Generate change events for the in predicate:
56.            if there is a hyperedge h such that
                 (h.trigger.isEmpty) and (h.guard.condition contains in(s)) then
57.                create change event for in(s) and add it to Q;
58.            fi
59.          od
```

If a state node s is entered that is mentioned as change event in(s) for some hyperedge, a change event for in(s) is generated.

```
4.6 Update F and C:
60.          F:=newF;
61.          C:=newC;
62.        fi
63.      fi
64.    od
```

Finally, the bag F of finished (terminated and completed) state nodes and the configuration C are updated. This cannot be done earlier since in lines 27, 36 and 49 we refer to the old values of C and F.

```
proc ComputeEnabled(C,event,F) ≡
 1.  foreach h ∈ rel(C,F) do // for every relevant hyperedge
 2.    if (h.trigger.notEmpty) then
 3.      if (event.oclIsTypeOf(ExternalEvent)) then
 4.        if (h.trigger.name=event.name) then
 5.          compute the truth value of the guard;
 6.          if the guard is true then enabled:=enabled + bag {h} fi
 7.        fi
 8.      elsif (event.oclIsTypeOf(TemporalEvent)) then
 9.        if (h=event.edge) then
10.          compute the truth value of the guard;
11.          if the guard is true then enabled:=enabled + bag {h} fi
12.        fi
13.     fi
14.   else // (h.trigger.isEmpty)
15.     if (h.guard=true) then enabled:=enabled + bag {h}
16.     elsif (event.type=change) then
17.       if (h.guard.condition ⇒ event.guard.condition) then
18.         enabled:=enabled + bag {h }
19.       fi
20.     elsif (event.type=termination) then
21.       if (event.activityinstance.node ∈ hyperedge.source) then
22.         compute the truth value of the guard
23.         if the guard is true then enabled:=enabled + bag {h} fi
24.       fi
25.     fi
26.   fi
27. od
```

Fig. 10. Procedure ComputeEnabled

Compute Enabled Hyperedges. The definition of procedure ComputeEnabled is given in Fig. 10. Each hyperedge in the bag of relevant hyperedges $rel(C, F)$ is tested whether it can become enabled. A hyperedge with a trigger event or temporal event becomes enabled iff its trigger event or temporal event occurs and the hyperedge's guard expression is true (2-13). Remember that we require that a hyperedge does not have both a trigger event and a temporal event. A hyperedge that has no trigger event becomes enabled iff its guard is true by definition (15), or if the current event is a change event that implies the hyperedge's guard condition (16-19), or if the current event is a termination event of one of the source state nodes of the hyperedge and the hyperedge's guard expression is true (20-24). Note that during guard evaluation the database may have to be accessed in order to find the current value of a variable (cf. Fig. 2).

Example. We give an example how a case of the Processing complaints workflow of Fig. 1 might be routed. Assume the current configuration contains action state nodes Register and Evaluate. Table 1 shows part of one possible execution

Table 1. Possible execution scenario for Processing Complaints (Fig. 1)

state	C	Q	**router**
1	{Register,Evaluate}		
2	{Register,Evaluate}		*Register*
3	{Register,Evaluate}	*Evaluate*	*Register*
4	{Send questionnaire, Evaluate}		*Evaluate*
5	{Send questionnaire, Evaluate}	*Send questionnaire*	*Evaluate*
6	{Send questionnaire, WAIT-3}	*Send questionnaire*	*WAIT-3*
7	{Send questionnaire, WAIT-3}		*Send questionnaire*
8	{WAIT-1,WAIT-3}		*WAIT-1*
9	{WAIT-1,WAIT-3}		
10	{WAIT-1,WAIT-3}		*questionnaire received*
11	{Process questionnaire, WAIT-3}		
12	{Process questionnaire, WAIT-3}		*Process questionnaire*
13	{WAIT-2,WAIT-3}	*in*(*WAIT-2*)	*WAIT-2*
14	{WAIT-2,WAIT-3}		*in*(*WAIT-2*)
15	{WAIT-2,Process complaint}		

scenario for this case by listing the consecutive states of the WFS. Due to space limitations we only show the first 15 states and we do not show the I and TL variables. We assume that the event manager puts events in I immediately in Q. All events, including completion and termination events, are written in *italic* font, while state nodes are listed in sans serif. If an event occurs and the router is not busy, we assume the router immediately starts processing this event (states 2, 10, and 12). We assume that completion events have priority over non-completion events.

5 Related Work

Although activity graphs are widely used for process modelling (see e.g. [5,13]), none of these references make any comments upon the semantics they attach to an activity graph.

The most important difference of our work with the OMG UML 1.3 semantics [14] is that in our semantics an activity is done in a state and by the environment, rather than in a transition by the system itself (see the introduction). UML CASE tools such as Rhapsody [10] implement the OMG semantics of activity graphs. Other formalisations of UML activity graphs [3,4] follow the OMG semantics very closely and they too map activities into transitions done by the system. Besides, these formalisations neither deal with real time nor treat events as objects.

Lilius and Paltor [12] have defined an execution algorithm for UML state machines. They only deal with named external events and do not treat temporal events. Moreover, they focus on run-to-completion steps, which are not relevant for activity graphs, since in activity graphs no call actions on transitions are used.

In [7] we presented a formal high-level semantics for activity graphs in which we assumed that routing does not take time. There, we did not give an execution algorithm but instead we defined a mathematical structure. Our present definition stays closer to both the original UML definition and the way WFSs are implemented in practice.

6 Conclusion

We presented an execution algorithm for UML activity graphs that is intended for workflow systems. Our algorithm stays close to the UML semantics of state machines but differs from it, in particular since run-to-completion is not relevant for activity graphs.

We are currently using this algorithm to verify activity graphs by model checking tools. With the execution algorithm the activity graph is mapped into a transition system which is the standard format for most model checkers. Next, we plan to investigate how object flows can be dealt with in the execution algorithm.

References

1. W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proc. CoopIS 2000*, LNCS 1901. Springer, 2000.
3. C. Bolton and J. Davies. Activity graphs and processes. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proc. IFM 2000*, LNCS 1945. Springer, 2000.
4. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. AMAST 2000*, LNCS 1826. Springer, 2000.
5. H.-E. Eriksson and M. Penker. *Business Modeling With UML: Business Patterns at Work*. Wiley Computer Publishing, 2000.
6. R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001.
7. R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In H. Hussmann, editor, *Proc. FASE 2001*, LNCS 2029. Springer, 2001.
8. P. Grefen and R. Remmerts de Vries. A reference architecture for workflow management systems. *Journal of Data & Knowledge Engineering*, 27(1):31–57, 1998.
9. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
10. I-Logix. Rhapsody. `http://www.ilogix.com`.
11. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 2000.
12. J. Lilius and I. Porres Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *Proc. UML'99*, LNCS 1723. Springer, 1999.
13. B. Paech. On the role of activity diagrams in UML. In Jean Bézivin and Pierre-Alain Muller, editors, *Proc. UML'98*, LNCS 1618. Springer, 1999.
14. UML Revision Taskforce. *OMG UML Specification v. 1.3*. Object Management Group, 1999.
15. Workflow Management Coalition. The workflow reference model (WFMC-TC-1003), 1995. `http://www.wfmc.org`.