A Real-Time Execution Semantics for UML Activity Diagrams

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 define a formal execution semantics for UML activity diagrams that is appropriate for workflow modelling. Our semantics is aimed at the requirements level by assuming that software state changes do not take time. It is based upon the STATEMATE semantics of statecharts, extended with some transactional properties to deal with data manipulation. Our semantics also deals with real-time and multiple state instances. We first give an informal description of our semantics and then formalise this in terms of transition systems.

1 Introduction

A *workflow* is a set of business activities that are ordered according to a set of procedural rules to deliver a service. A workflow model (or workflow specification) is the definition of a workflow. An instance of a workflow is called a *case*. Examples of cases are an insurance claim handling instance and a production order handling instance. The definition, creation, and management of workflow instances is done by a workflow management system (WFMS), on the basis of workflow models. We represent workflow models by UML activity diagrams [16].

In this paper, we define a formal execution semantics for UML activity diagrams that is suitable for workflow modelling. The goal of the semantics is to support execution of workflow models and analysis of the functional requirements that these models satisfy. Our long term goal is to implement the execution semantics in the TCM case tool [6] and to use model checking tools for the analysis of functional requirements. A secondary goal of the semantics is to facilitate a comparison with other formal modelling techniques for workflows, like Petri nets [1] and statecharts [18]. For example, some people claim that an activity diagram is a Petri net. But in order to sustain this claim, first a formal semantics for activity diagrams must be defined, so that the Petri net semantics and the activity diagram semantics can be compared.

Figure 1 shows an example activity diagram. Ovals represent activity states, rounded rectangles represent wait states, and arrows represent state transitions. Section 3 explains the details of the notation. We use activity diagrams to model a single instance (case) of a workflow. We defer the modelling of multiple cases (case management) to future work.

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

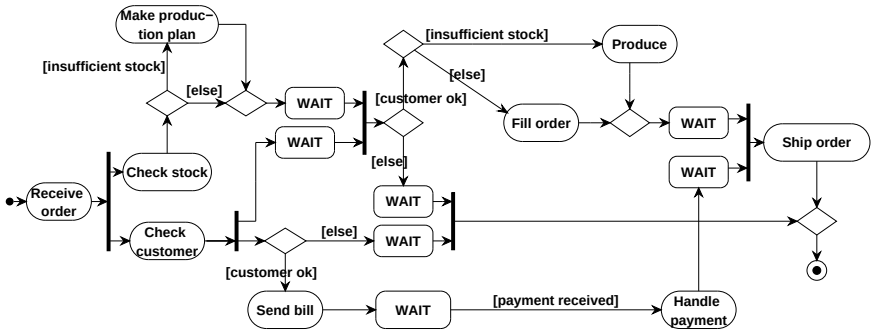

Fig. 1. Example activity diagram

We introduce two semantics. The first semantics supports execution of workflow models. Although this semantics is sufficient for executing workflow models, it is not precise enough for the analysis of functional requirements (model checking), since the behaviour of the environment is not formalised. We therefore define a second semantics, which we will use for model checking, that extends the first one by formalising the combined behaviour of both the system that the activity diagram models and the system's environment.

Our semantics is different from the OMG activity diagram semantics [16], because we map activities into states, whereas the OMG maps them into transitions. The OMG semantics implies that activities are done by the WFS itself, and not by the environment. In our semantics, activities are done by the environment (i.e. actors), not by the WFS itself (see Sect. 2 for our motivation).

The paper is structured as follows. In Sect. 2 we discuss workflow concepts and we give an informal semantics of activity diagrams in terms of the domain of workflow. In Sect. 3 we define the syntax of an activity diagram, and define and discuss constraints on the syntax. In Sect. 4 we define our two formal semantics. In Sect. 5 we briefly discuss other formalisations of activity diagrams. We end with a summary and a discussion of further work. Formulas are written in the Z notation [17].

2 Workflow Domain

Workflow concepts. The following exposition is based on literature (amongst others [1, 14]) and several case studies that we did.

Activities are done by *actors*. Actors are people or machines. An *activity* is an uninterruptible amount of work that is performed in a non-zero span of time by an actor. In an activity, *case attributes* are updated. Case attributes are data relevant for the case. They may be present in the form of structured data, or case documents. The *effect* of an activity is constrained declaratively with a pre and post-condition. The pre-condition also functions as guard: as long as it is false, the activity cannot be performed.

The case may be distributed over several actors. Each distributed part of the case has a *local state*. There are three kinds of possible local states.

- In an *activity state* an actor is executing an activity in a part of the case. For every activity there should be at least one activity state, but different activity states can represent execution of the same activity.
- In a *wait state*, the case is waiting for some external event or temporal event.
- In a *queue state*, the case is waiting for an actor to become available to perform the next activity of the case.

Multiple instances of a local state may be active at the same time in the same case. The *global state* of the case is therefore a multiset (rather than a set) of the local states of the distributed parts of the case.

The WFMC [19] specifies four possible ordering relationships between activities: *sequence*, *choice*, *parallelism* and *iteration*. And to facilitate readability and re-use of process definitions, an ordered set of activities can be grouped into one *compound activity*. A compound activity can be used in other process definitions. A non-compound activity is called an *atomic activity*.

System structure (Fig. 2). A workflow system (WFS), which is a WFMS instantiated with one or more workflow models, connects a database and several actors. Since we use an activity diagram to model a single case, we here assume the WFS controls a single case. The WFS routes the case as prescribed by the workflow model of the case. Note that the case attributes are updated during an activity by the actors, not by the WFS. For example, an actor may update a claim form by editing it with a word processor. The state of the case, on the other hand, is updated by the WFS, not by an actor. All attributes of a case are stored in the database. The state of the case is maintained by the WFS itself.

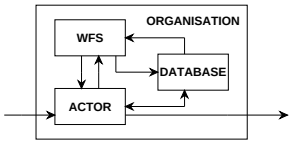

Fig. 2. System structure

Informal semantics. An activity diagram is a requirements specification that says what a WFS should do. We therefore define our semantics of an activity diagram in terms of a WFS.

We view a WFS as a reactive system. A reactive system [12] is a system that runs in parallel with its environment, and reacts to the occurrences of certain events in its environment by creating certain desirable effects in that environment. There are three kinds of events:

- An *external event* is an instantaneous, 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.
  - A *value change event* is an event that represents change of one or more variables.

– A *temporal event* is a moment in time, to which the system is expected to respond, i.e. some deadline has been reached.

The behaviour of a reactive system is modelled as a set of runs. A *run* is a sequence of system states and system reactions. System reactions are caused by the occurrence of events.

We make the following two assumptions. First, the goal of our semantics is to support the specification of functional requirements. Functional requirements should be specified independently of the implementation platform, and we therefore make the *perfect technology* assumption: the implementation consists of infinitely many resources that are infinitely fast [15]. Perfect technology implies that the WFS responds infinitely fast to events. This means that the transitions between the local states of a case take no time. But since actors are not assumed to be perfect, they do take time to perform their activities. Second, the WFS responds as soon as it receives events from the environment. This is called the *clock-asynchronous semantics* [11]. These two assumptions together imply that the WFS responds at the same time events occur. This is called the *perfect synchrony hypothesis* [3].

For an implementation of the WFS, these two assumptions translate into the following requirement:

> the WFS must be fast enough in its reaction to the current events to be ready before the next events occur

For runs, these two assumptions imply that time elapses in states only, not in reactions, since reactions are instantaneous, and that there elapses no time between the occurrence of events and the subsequent reaction of the system.

The state of a run of the WFS consists of the following components:

– the global state of the case, including an indication of which activities are currently being performed,
– the current set of input events,
– the current value of the case attributes of the case,
– the current value of the running timers. These are necessary to generate time-outs.

The global state of the case is the multiset of the local states of the individual parallel branches that are active. We call such a global state a *configuration*. Each parallel branch has three kinds of possible states, namely activity, wait and queue states. For the remainder of this paper, we do not consider queue states anymore; we simply assume that there enough actors available for every activity. This is not a serious restriction, because a queue state can be modelled as a wait state, where the event that has to be waited for is that the actor becomes available.

During a reaction, the state of the case is updated (i.e. the case is routed), some timers may be reset, and the set of input events is reset, but the case attributes are not changed. Case attributes are updated by actors during an activity state. During a reaction the current time and the timers do not increase, because a reaction is instantaneous.

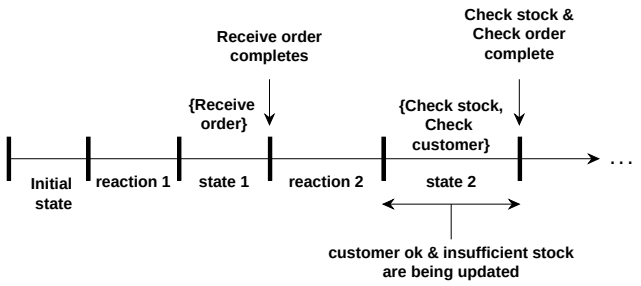

Fig. 3. Run of our example. In each state, the set of activities currently executing is shown.

We adopt the STATEMATE [11] semantics of a reaction and extend it below with some transactional properties. Before the events occur, the system is in a stable state. When the events occur, the system state has become unstable. To reach a stable state again, the system *reacts* by taking a *step* and entering a new state. If the new state is unstable, again a step is taken, otherwise the system stops taking steps. This sequence of taking a step and entering a new state and testing whether the new state is stable, is repeated until a stable state is reached. Thus, the system reaction is a sequence of steps, called a *superstep* [11]. In Sect. 4 we define steps and supersteps. For the initial state, we do not have to wait for a change in the environment since the initial state is unstable.

Next, we adopt the following assumptions from STATEMATE [11]:

- More than one event can be input to the system at the same time.
- The input is a set of events, rather than a queue (the latter assumption is adopted by the OMG semantics [16], but is more appropriate for a software implementation-level semantics).
- If the system reacts, it reacts to all events in the input set.
- Events live for the duration of one step only (not a superstep!).

Figure 3 shows part of a run of our example. In each state of the run the set of currently executing activities is shown. Both customer ok and insufficient stock are case attributes. See Sect. 4 for details on how a run is constructed.

Specifying activities. An activity has a pre and post-condition. The pre-condition specifies when the activity is allowed to start. The post-condition specifies constraints on the result of the activity. Both pre and post-conditions only refer to case attributes. A case attribute can either be *observed* in an activity by an actor, i.e., used but not changed, or *updated* in an activity by an actor. This may result in an ill-defined case attribute, if the attributes is accessed in two or more concurrently executing activities, and in addition one of the activities updates it. Then the activities *interfere* with each other. Non-interference checks can prevent this. We define an non-interference check on activities, since we view an activity as atomic. An activity may consist of many database transactions (cf. Fig. 2), so non-interference of data manipulation in an activity cannot be

handled by a single transaction of the database. Instead, we assume that non-interference checks are done by the WFS (for example by means of a transaction processing monitor that is part of the WFS).

3 Syntax of Activity Diagrams

A UML activity diagram is a graph, consisting of state nodes and directed edges between these state nodes. There are two kinds of state nodes: ordinary state nodes and pseudo state nodes. We follow the UML in considering pseudo state nodes as syntactic sugar to denote hyperedges. Thus, the underlying syntactic structure is a hypergraph, rather than a graph. In order to be unambiguous, we call the underlying hypergraph an *activity machine*.

UML constructs used (Fig. 4). We use an action state node to denote an activity state, a wait state node to denote a wait state, and a subactivity state node to denote a compound activity state. We assume that for every compound activity state node, there is an activity diagram that specifies the behaviour of the compound activity. The transitive closure of this hierarchy relation between activity diagrams must be acyclic. Besides these state constructs, we use pseudo state nodes to indicate xor split and merge (decision state node, merge state node), parallelism (fork state node, join state node), begin (initial state node) and end (final state node). 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]. State nodes (including pseudo state nodes) are linked by directed, labelled edges (expressing sequence). Each label has the form $e[g]$ where e is an event expression and g a guard expression. Empty event NULL and guard [true] are not shown on an edge. Special event label after(*texp*) denotes a relative temporal event, where *texp* is an integer expression. See our report [8] for other temporal constructs (e.g. when).

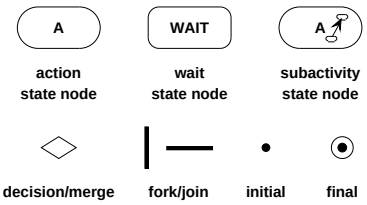

Fig. 4. UML activity diagram constructs used

UML constructs removed.

- We do not label edges with action expressions. Actions are performed by actors in activities, not by the WFS in the transitions of a workflow.
- We do not consider synch (synchronisation) states. We have never seen an example of a synch state in our own or other people's case studies.
- We do not consider deferred events (deferring an event means postponing the response to an event). Deferral of event e can be simulated by using the guard [e occurred].
- Swimlanes allocate activities to packages, actors, or organisational units. We disregard swimlanes, since these do not impact the execution semantics. We plan to consider allocation of activities to actors at a later stage.

- The UML includes object flow states, that denote data states. They are connected to other state nodes by object flows (dashed edges). There are several ambiguities concerning object flow states, the most important one being that the meaning of parallel object flow states is not defined. Besides, only one vendor of workflow management systems supports object flow states [14]. For the moment, we decide to omit object flow states (and thus object flows) from our syntax. Instead, we represent the case attributes by the local variables of the activity diagram and assume these attributes are stored in a database.
- Dynamic concurrency (i.e. dynamic instantiation of multiple instances of the same action or subactivity state node) we treat in our full report [8].

Syntax of activity machines. An activity machine is a rooted directed hypergraph. Figure 5 shows the activity machine corresponding to Fig. 1. We assume given a set *Activities* of activities. An *activity machine* is a quintuple $(Nodes, Edges, Events, Guards, LVar)$ where:

- $Nodes = AS \cup WS \cup \{initial, final\}$ is the set of state nodes,
- $Edges \subseteq \mathbb{P}\, Nodes \times Events \times Guards \times \mathbb{P}\, Nodes$ is the transition relation between the state nodes of the activity diagram,
- *Events* is the set of external event expressions,
- *Guards* is the set of guard expressions,
- *LVar* is the set of local variables. The local variables represent the case attributes. We assume that every variable in a guard expression is a local variable.

State nodes *initial* and *final* denote the initial and final state node, respectively. Besides these special state nodes, an activity machine has action state nodes AS and wait state nodes WS. Every action state node has an associated activity it controls, denoted by the surjective function $control : AS \twoheadrightarrow Activities$. The execution of the activities falls outside the scope of the activity machine, since it is done by actors. We use the convention that in the activity diagram, an action state node a is labelled with the activity $control(a)$ it controls. Note that different action state nodes may bear the same label, since they may control the same activity. Wait state nodes are labelled WAIT. Edges are labelled with events and guard expressions out of sets *Events* and *Guards* respectively. A special element of *Events* is the empty event NULL, which is always part of the input. Given $e = (N, ev, g, N') \in Edges$, we define $source(e) \stackrel{\text{df}}{=} N$, $event(e) \stackrel{\text{df}}{=} ev$, $guard(e) \stackrel{\text{df}}{=} g$, and $target(e) \stackrel{\text{df}}{=} N'$.

We require that the initial state node only occurs in the source of an edge. Moreover, if it is source of an edge, it is the only source of that edge. Similarly, the final state node may only occur in the target of an edge. Moreover, if it is target of an edge, it is the only target of that edge. Next, the initial state must be unstable. Therefore, the edges leaving the initial state node must be labelled with the empty event NULL and the disjunction of the edges' guard expressions must be a tautology.

Let set $BE(LVar)$ denote the set of boolean expressions on set $LVar$. Next we define the infinite set of all possible $Timers \stackrel{\text{df}}{=} \{t(e)(n) \mid e \in Edges \wedge n \in \mathbb{N}\}$.

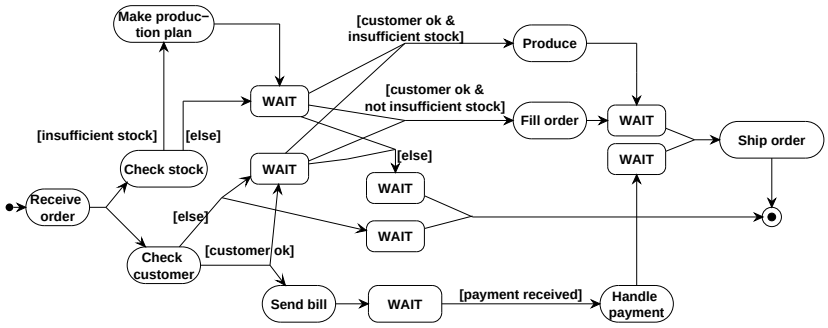

Fig. 5. Activity machine of our running example

Roughly, a timer $t(e)(n)$ is reset to zero every time the source states of e are entered. We assume a set $BE(Timers)$ of basic clock constraints on timers. Every basic clock constraint $\phi \in BE(Timers)$ has the form $c = texp$ where $c \in Timers$ and $texp \in \mathbb{N}$.

Set *Guards* is constructed as the union of $BE(LVar)$ and $BE(Timers)$ and the set of expressions that is obtained by conjoining ($\wedge$) elements of the sets $BE(LVar)$ and $BE(Timers)$. We require that if $t(e)(n) = texp$ is part of the guard expression of edge e', then $e = e'$.

Mapping an activity diagram to an activity machine. In the mapping, first the subactivity state nodes are eliminated by substituting for every subactivity state node its corresponding activity diagram. Then the pseudo state nodes are removed and replaced by hyperedges. Finally, every hyperedge e with label $\mathsf{after}(texp)$ is replaced by infinitely many hyperedges $e(n)$ where $n \in \mathbb{N}$, each edge labelled with clock constraint $t(e)(n) = texp$. (This construction is needed, because a finite but unbounded number of instances of e may be taken simultaneously at the same time.) Our full report [8] gives more details.

Specifying data manipulation in activities. The local variables of the activity diagram are possibly updated in activities (since local variables represent case attributes). In every activity $a \in Activities$ that is controlled by an activity diagram, some local variables may be *observed* or *updated*. We denote the observed variables by $Obs(a) \subseteq LVar$, and the updated variables by $Upd(a) \subseteq LVar$. We require these two sets to be disjoint for each activity.

Two activities *interfere* with each other, if one of them observes or updates a local variable that the other one is updating. (This definition is similar to the definition of conflict equivalence in database theory [7].) Note that in particular every activity only non-interferes with itself iff it only observes variables.

$$A \lightning B \Leftrightarrow (Obs(A) \cup Upd(A)) \cap Upd(B) \neq \varnothing \\ \vee\ (Obs(B) \cup Upd(B)) \cap Upd(A) \neq \varnothing$$

Furthermore, we define for every activity a a pre and post-condition, pre(a) and post(a). The precondition only refers to variables in $Obs(a) \cup Upd(a)$. The post-condition only refers to variables in $Upd(a)$, since the observed variables are not changed.

We assume a typed data domain $\mathcal{D}$. Let $\sigma : LVar \to \mathcal{D}$ be a total, type-preserving function assigning to every local variable a value. We call such a function a *valuation*. Let $\Sigma(LVar)$ denote the set of all valuations on $LVar$. A partial valuation is a valuation that is a partial function. The set of all partial valuations we denote by $\Sigma_{\mathrm{p}}(LVar)$. A pre or post-condition c always is evaluated w.r.t. some (partial) valuation σ. We write $\sigma \models c$ if c is true in σ (with for every variable v its value $\sigma(v)$ substituted). We do not define the syntax of c: this depends on the data types used. Since we have defined no formal syntax for pre and post-conditions, we do not provide a formal semantics for the satisfaction relation $\models$. In our semantics we simply assume that a formal syntax and semantics for pre and post-conditions has been chosen.

Function $\mathit{effect} : \Sigma_{\mathrm{p}}(LVar) \times AS \nrightarrow \mathbb{P}\,\Sigma(LVar) - \{\varnothing\}$ is a partial function constraining the possible effects of each activity on the case attributes. For a given activity a and partial valuation $\sigma \in \Sigma(Obs(a) \cup Upd(a))$, the set of possible valuations is $\mathit{effect}(\sigma, a) = \{\sigma' \mid \sigma' \models \mathit{post}(a) \wedge \forall\, v \in Obs(a) \bullet \sigma(v) = \sigma'(v)\}$.

Since we allow multiple instances of an activity to be executing, we work with multisets of activities. Given a multiset of activities A that do not interfere, if $\mathit{effect}(\sigma, a)$ is a possible effect of activity a, the combined effect of activities in A is $\uplus_{a \sqsubset A} \mathit{effect}(\sigma, a)$, where $\sqsubset$ is the membership predicate for multisets and $\uplus$ is union on multisets. Due to the non-interference constraint, the only overlap can be in observed variables and these remain unchanged. We denote the set of possible combined effects with $\mathit{effect}(\sigma, A)$.

Finally, we lift all functions with domain *Activities* to the domain of action state nodes AS by means of function $\mathit{control} : AS \twoheadrightarrow \mathit{Activities}$, defined above. For example, for $a \in AS$, $\mathit{effect}(\sigma, a) \stackrel{\mathrm{df}}{=} \mathit{effect}(\sigma, \mathit{control}(a))$.

Data-related syntactic constraints (Fig. 6). First, every action state node must be followed by a subsequent wait state node. Our semantics will ensure that this wait state node is left iff all variables that have to be tested are not being updated anymore. This prevents a case from being illegally routed. Second, we do not consider pre-conditions in our semantics, since for every activity A that is followed by an activity B if guard g is true, we have that [g] ⇒ post(A) and post(A) ⇒ pre(B). We conclude [g] ⇒ pre(B). So we drop the precondition of an activity from our semantics, since it is already implied by the preceding guard.

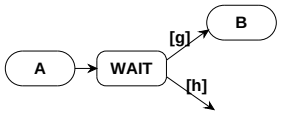

Fig. 6. Modelling pre and post-conditions

4 Execution Semantics of Activity Machines

We give two formal semantics for activity machines that are based upon the informal semantics of Sect. 2. The first semantics defines how a step is computed.

This semantics can be used to execute an activity machine, but is not a complete definition of a run. The second semantics, useful for model checking, extends the first one by defining a transition system, whose execution paths are runs. The transition system we use is a clocked Kripke structure. Both semantics are an adaptation of the semantics we defined earlier for UML statecharts [9].

4.1 Computing a Step

States. At each point in time, a system state consists of the current configuration C, the current input I, the current value of every local variable $v \in \mathit{LVar}$, and the valuations of the set $\mathit{On} \subseteq \mathit{Timers}$ of running timers. So the system state is a valuation $\sigma : \{C, I\} \cup \mathit{LVar} \cup \mathit{On} \to \mathcal{D}$.

The configuration is a multiset of state nodes $\mathit{Nodes} \to \mathbb{N}$ of the activity machine. A configuration is non-interfering, written $\mathit{non\text{-}interfering}(C)$, iff it does not contain interfering action state nodes:

$$\mathit{non\text{-}interfering}(C) \stackrel{\text{df}}{\Leftrightarrow} \forall\, a, a' \sqsubseteq C \bullet \neg\, (a \lightning a')$$

where as before $\sqsubseteq$ denotes the membership predicate for multisets. In the sequel, the only configurations we allow are non-interfering ones.

Input. We define input I to be a tuple $(\mathit{Ev}, \sigma_{\mathrm{p}}^{\mathit{LV}}, T, \sigma_{\mathrm{p}}^{\mathrm{t}}) \in \mathit{Events} \times \Sigma_{\mathrm{p}}(\mathit{LVar}) \times (\mathit{AS} \to \mathbb{N}) \times \Sigma_{\mathrm{p}}(\mathit{Timers})$, where as before $\Sigma(S)$ denotes the set of all valuations on set S. Set Ev is the set of external events. We require that empty event NULL is always input: $\{\mathit{NULL}\} \subseteq \mathit{Ev}$. Partial valuation $\sigma_{\mathrm{p}}^{\mathit{LV}}$ represents the set of value change events. The partial valuation assigns to every local variable that is changed its new value. A special value change event occurs when an action state node has terminated because its corresponding activity has completed. This event is modelled by T, which denotes the multiset of terminated action state nodes. We require that only action state nodes in the configuration can terminate: $T \sqsubseteq (\mathit{AS} \lhd C)$, where $\sqsubseteq$ denotes the sub-multiset relation and $X \lhd Y$ denotes restriction of relation Y to the domain set X. Finally, partial valuation $\sigma_{\mathrm{p}}^{\mathrm{t}}$ represents the set of temporal events. A temporal event occurs because some running timer has reached a certain desired value. We therefore require that every timer in the domain of $\sigma_{\mathrm{p}}^{\mathrm{t}}$ is running: $(\mathit{On} \lhd \sigma_{\mathrm{p}}^{\mathrm{t}}) \subseteq \sigma_{\mathrm{p}}^{\mathrm{t}}$.

Steps. A step is a maximal, consistent sub-multiset of the multiset of enabled edges. In addition, the new configuration must be non-interfering. Our definitions extend and generalise both the STATEMATE semantics [11] and our semantics for UML statecharts [9] from sets of states (edges) to multisets of states (edges).

Before we define the multiset of enabled edges $\mathit{En}(C, \mathit{Ev}, T)$, we observe that an edge leaving an action state node a is only enabled if a has terminated, since otherwise the corresponding activity would not be atomic. Therefore, in the definition we do not consider the current configuration C, but instead the multiset C' of non-action state nodes in the current configuration joined with the multiset T of terminated action state nodes. An edge e is n times enabled

in the current state σ of the system iff $source(e)$ is contained in C', one of the input events is $event(e)$, the guard can be safely evaluated (denoted by predicate $eval$) and moreover evaluates to true (denoted $\models$) given the current values of all variables, and n is the minimum number of instances of the source state nodes that can be left. Predicate $eval$ states that a guard g can be safely evaluated iff it does not refer to variables that are being updated in the current valuation σ in some activities. We do not refer to σ_{p}^{LV} and $\sigma_{\mathrm{p}}^{\mathrm{t}}$ because these are contained in σ. In formulas:

$$\begin{aligned} En(C, Ev, T) \stackrel{\mathrm{df}}{=} \{ e \mapsto n \mid\ & ms(source(e)) \sqsubseteq C' \wedge event(e) \in Ev \\ & \wedge\ eval(\sigma, guard(e)) \wedge \sigma \models guard(e) \\ & \wedge\ n = min(\{ C' \sharp s \mid s \in source(e) \}) \ \} \\ & \text{where } C' = (((Nodes - AS) \lhd C) \uplus T) \end{aligned}$$

$$eval(\sigma, g) \quad \stackrel{\mathrm{df}}{\Leftrightarrow} \forall\, a \sqsubseteq AS \lhd (C \sqcup T) \bullet var(g) \cap Upd(a) = \varnothing$$

where $\uplus$ denotes multiset union, $M \sharp x$ is the number of times x appears in multiset M, $ms(S) \stackrel{\mathrm{df}}{=} \{s \mapsto 1 \mid s \in S\}$ (this coerces a set into a multiset), $\sqcup$ is difference on multisets, and $var(g)$ denote the set of variables that g tests.

Given configuration C, a multiset of edges E is defined to be consistent, written $consistent(C, E)$, iff all edges can be taken at the same time, i.e. taking one does not disable another one:

$$consistent(C, E) \stackrel{\mathrm{df}}{\Leftrightarrow} (\uplus_{e \sqsubset E} ms(source(e))) \sqsubseteq C$$

The function $nextconfig$ returns the next configuration, given a configuration C and a consistent multiset of edges E:

$$nextconfig(C, E) \stackrel{\mathrm{df}}{=} C \sqcup \biguplus_{e \sqsubset E} ms(source(e)) \ \uplus\ \biguplus_{e \sqsubset E} ms(target(e))$$

Below, we require that taking a step leads to a non-interfering new configuration.

A multiset of edges E is defined to be maximal iff for every enabled edge e that is added to E, multiset $E \uplus [\![e]\!]$ is inconsistent or the resulting configuration is interfering. Notation $[\![e]\!]$ denotes a bag that contains e only.

$$\begin{aligned} maximal(C, Ev, T) \stackrel{\mathrm{df}}{\Leftrightarrow} \forall\, e \sqsubseteq En(C, Ev, T) \mid e \not\sqsubseteq E \bullet \neg\ & consistent(C, E \uplus [\![e]\!]) \\ & \vee \neg\ non\text{-}interfering(nextconfig(C, E)) \end{aligned}$$

Finally, predicate $isStep$ defines a multiset of edges E to be a step iff every edge in E is enabled, E is maximal and consistent, and the next configuration is noninterfering. In our full report [8], this definition is written out as a step algorithm. In the next subsection, we will use the predicate $isStep$ again.

$$\begin{aligned} isStep(E) \stackrel{\mathrm{df}}{\Leftrightarrow}\ & E \sqsubseteq En(C, Ev, T) \wedge consistent(C, E) \\ & \wedge\ maximal(C, Ev, T) \wedge non\text{-}interfering(nextconfig(C, E)) \end{aligned}$$

4.2 Transition System Semantics of Activity Machines

A Clocked Kripke Structure (CKS) is a quadruple $(Var, \rightarrow, ci, \sigma_0)$ where:

- $Var = \{C, Ev, T\} \cup LVar \cup On$ is the set of variables,
- $\rightarrow \subseteq \Sigma(Var) \times \Sigma(Var)$ is the transition relation,
- ci is the clock invariant, a constraint that must hold in every valuation,
- $\sigma_0 \in \Sigma(Var)$ is the initial valuation.

We have omitted from Var the I components σ_{p}^{LV} and $\sigma_{\mathrm{p}}^{\mathrm{t}}$ since these are already modelled by $LVar$ and On respectively.

Given an activity machine, its CKS is constructed as follows. First, we specify the clock invariant. For every basic clock constraint $t(e)(n) = texp$, we specify a constraint ϕ of the form $t(e)(n) \in On \Rightarrow t(e)(n) \leq texp$. Clock invariant ci is the conjunction of all constraints ϕ. We evaluate a clock invariant ci in a valuation σ, and write $\sigma \models ci$ if the clock invariant is true.

The transition relation $\rightarrow$ is specified as the union of three other transition relations. Not every sequence of transitions out of this union satisfies the clock-asynchronous semantics. Every valid sequence must start with a superstep (the initial step) followed by a sequence of cycles. The initial superstep is taken because the initial state is by definition unstable. A cycle is one or more time steps, followed by an event step, followed by a superstep. Note that $\rightarrow_{cycle}$ is not part of $\rightarrow$.

$$\rightarrow_{cycle} \stackrel{\mathrm{df}}{=} \rightarrow_{timestep}^{+} \fatsemi \rightarrow_{event} \fatsemi \rightarrow_{superstep}$$

Relation $\rightarrow_{timestep}$ represents the elapsing of time by updating timers with a delay Δ such that the clock invariant is not violated. In the following, $\&_{s \in S}$ denotes a concurrent update done for all elements of set S.

$$\begin{aligned}\sigma \rightarrow_{timestep} \sigma' \stackrel{\mathrm{df}}{\Leftrightarrow} \exists\, \Delta \in \mathbb{R} \mid \Delta > 0 \bullet \sigma' = \sigma[\&_{c \in On}\, c/\sigma(c) + \Delta] \\ \text{such that } \forall\, \delta \in [0, \Delta] \bullet \sigma[\&_{c \in On}\, c/\sigma(c) + \delta] \models ci\end{aligned}$$

Relation $\rightarrow_{event}$ defines that events occur between σ and σ' iff timers did not change, the configuration did not change, and the local variables that are being updated in activities are not changed, and:

- either there are named external events in the input in state σ';
- or there is a nonempty set L of local variables that have no interference with the currently executing activities and whose value changed;
- or some action state nodes have terminated, and there is a partial valuation $\sigma'_{\mathrm{p}} \subset \sigma'$ that conforms to the effect constraints of the currently terminating activities;
- or it is not possible to do any more time steps (i.e. a deadline is reached).

$$\begin{aligned}\sigma \rightarrow_{event} \sigma' \stackrel{\mathrm{df}}{\Leftrightarrow} (\forall\, c \in On \bullet \sigma(c) = \sigma'(c)) \wedge \sigma(C) = \sigma'(C) \\ \wedge\ \forall\, v \in LVar;\ \forall\, a \in AS \mid a \sqsubseteq \sigma(C) \uplus \sigma'(T) \bullet\end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad v \in Obs(a) \cup Upd(a) \Rightarrow \sigma(v) = \sigma(v') \\
\wedge\ (\quad \sigma'(Ev) \subseteq Events \wedge \{NULL\} \subset \sigma'(Ev) \\
\quad \vee\ \exists\, L \subseteq LVar \mid L \neq \varnothing \bullet \\
\qquad (\forall\, a \in AS \mid a \sqsubseteq \sigma(C) \bullet L \cap (Obs(a) \cup Upd(a)) = \varnothing) \\
\quad \vee\ [\![]\!] \neq \sigma'(T) \sqsubseteq (AS \lhd \sigma(C)) \wedge \exists\, \sigma'_{\mathrm{p}} \in \mathit{effect}(\sigma, \sigma'(T)) \bullet \sigma'_{\mathrm{p}} \subset \sigma' \\
\quad \vee\ \nexists \sigma'' \bullet \sigma \rightarrow_{timestep} \sigma'' \quad)
\end{array}
$$

Finally, a superstep is a sequence of steps, such that intermediate states are unstable and the final state of the sequence is stable. The notation $f \oplus g$ means that function g overrides function f on the domain of f. Note that the intermediary states (the semicolon in the composition of the relations) are not part of the CKS.

$$
\begin{array}{lll}
\rightarrow_{superstep} & \stackrel{\mathrm{df}}{=} (\rightarrow_{unstable} \fatsemi \rightarrow_{step} \fatsemi \rightarrow_{superstep}) \cup \rightarrow_{stable} & (1) \\
\sigma \rightarrow_{step} \sigma' & \stackrel{\mathrm{df}}{\Leftrightarrow} \exists\, E \mid \mathit{isStep}(E) \bullet & \\
 & \qquad \exists\, S_1 \subseteq Timers \mid \mathit{OffTimers}(\sigma(C), E, \sigma(On), S_1); & \\
 & \qquad \exists\, S_2 \subseteq Timers \mid \mathit{NewTimers}(\sigma(C), E, \sigma(On), S_2) \bullet & \\
 & \qquad\quad \sigma' = \sigma[C/\mathit{nextconfig}(\sigma(C), E), Ev/\varnothing, T/[\![]\!], & \\
 & \qquad\qquad \&_{s \in S_2} s/0, On/\sigma(On) - S_1 \cup S_2] & \\
\sigma \rightarrow_{unstable} \sigma' & \stackrel{\mathrm{df}}{\Leftrightarrow} \sigma = \sigma' \wedge \mathit{En}(\sigma(C), \sigma(Ev), \sigma(T)) \neq \varnothing & \\
\sigma \rightarrow_{stable} \sigma' & \stackrel{\mathrm{df}}{\Leftrightarrow} \sigma = \sigma' \wedge \mathit{En}(\sigma(C), \sigma(Ev), \sigma(T)) = \varnothing &
\end{array}
$$

Line by line, the $\rightarrow_{step}$ definition says that a step is done between σ and σ' iff:

- there is a step E (using the predicate *isStep* defined in Sect. 4.1);
- there is a set S_1 of timers that can be turned off (denoted by predicate *OffTimers*;
- there is a set S_2 of timers that can be turned on (denoted by predicate *NewTimers*;
- σ is then updated into σ' by computing the next configuration when step E is performed (using the function *nextconfig* defined in Sect. 4.1), and resetting the input, the multiset of terminated action state nodes, and all the new timers in S_2, and finally updating *On*.

Predicates $\rightarrow_{unstable}$ and $\rightarrow_{stable}$ test whether there are enabled edges. We compute a superstep by taking a least fixpoint of (1). This may not exist; in which case the superstep does not terminate. Or it may not be unique, in which case there is more than one possible superstep.

We now proceed to define predicates *OffTimers* and *NewTimers*. First we define the notion of relevant hyperedges. Given configuration C, the multiset of relevant hyperedges $\mathit{rel}(C)$ contains each edge whose source is contained in C.

$$
\mathit{rel}(C) = \{e \mapsto n \mid \mathit{source}(e) \subseteq C \wedge n = \mathit{min}(\{C \sharp s \mid s \in \mathit{source}(e)\})\}
$$

For every relevant edge with a clock constraint a timer is running. This timer was started when the edge became relevant. It will be stopped when the edge will

become irrelevant. Assume given a configuration C, a step E, a set of running timers On, and a set S of timers. Predicate *OffTimers* is true iff all timers in S are running, but can now be turned off, because their corresponding edges are relevant for C, but are no longer relevant if E is taken. Predicate *NewTimers* is true iff all timers in S are off, but can now be turned on, because their corresponding edges are irrelevant for C, but do become relevant if E is taken.

$$\begin{aligned}
\mathit{OffTimers}(C, E, On, S) \;&\Leftrightarrow S \subseteq On \cap \{t(e)(n) \mid e \sqsubseteq R \wedge n \in \mathbb{N}\} \\
&\wedge\ \forall\, e \sqsubseteq R \bullet R \sharp e = \#(S \cap \{t(e)(n) \mid n \in \mathbb{N}\}) \\
&\quad \text{where } R = \mathit{rel}(C) - \mathit{rel}(\mathit{nextconfig}(C, E)) \\
\mathit{NewTimers}(C, E, On, S) \;&\Leftrightarrow S \subseteq (\mathit{Timers} - On) \cap \{t(e)(n) \mid e \sqsubseteq R \wedge n \in \mathbb{N}\} \\
&\wedge\ \forall\, e \sqsubseteq R \bullet R \sharp e = \#(S \cap \{t(e)(n) \mid n \in \mathbb{N}\}) \\
&\quad \text{where } R = \mathit{rel}(\mathit{nextconfig}(C, E)) - \mathit{rel}(C)
\end{aligned}$$

In the initial valuation σ_0, the configuration only contains one copy of *initial*, the only input event is NULL, and On is empty.

5 Related Work

The OMG [16] gives a semantics to an activity diagram by translating it into a UML statechart. Both the translation and the semantics of UML statecharts are not formally defined. Moreover, the translation is inappropriate, since activity diagrams are more expressive than statecharts. The OMG semantics (and other semantics too [4, 5]) maps action state nodes to transitions. This means that updates to case attributes are made by the WFS itself, and not by the actors. Our semantics maps action state nodes to states (valuations), which means that activities are performed by actors, not by the WFS.

Gehrke *et al.* [10] give a semantics by translating an activity diagram into a Petri net. Their semantics does not deal with data or time as we do. In addition, Petri net semantics do not model the environment, whereas our semantics does model the environment. We provide a more detailed comparison with all these other formalisations in our full report [8].

6 Conclusions and Future Work

We have defined a formal real-time requirements-level execution semantics for UML activity diagrams that manipulate data, for the application domain of workflow modelling. The semantics is based on the STATEMATE statechart semantics, extended with transactional properties. We defined both an execution and a transition system semantics. Our semantics is different from other proposed semantics, both for activity diagrams and for workflow models. It is motivated by analysis of the workflow literature and by case studies.

We have done initial experiments with model checking simple statecharts using the model checker Kronos [13]. Future work includes extending this to model checking activity machines. Next, we plan a detailed comparison with Petri net semantics.

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. Workflow pattern home page: `http://www.mincom.com/mtrspirt/workflow`.
3. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comp. Prog.*, 19(2):87–152, 1992.
4. C. Bolton and J. Davies. Activity graphs and processes. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proc. Integrated Formal Methods 2000*, LNCS 1945. Springer, 2000.
5. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000*, LNCS 1826. Springer, 2000.
6. F. Dehne, R. Wieringa, and H. van de Zandschulp. Toolkit for conceptual modeling (TCM) — user's guide and reference. Technical report, University of Twente, 2000. `http://www.cs.utwente.nl/~tcm`.
7. R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamins/Cummings, Redwood City, 1989.
8. R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams, 2001. Available at `http://www.cs.utwente.nl/~eshuis/adsem.ps`.
9. R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*, pages 121–140. Kluwer Academic Publishers, 2000.
10. T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. Hildesheimer Informatik-Bericht 11/98, 1998.
11. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
12. D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO/ASI 13. Springer, 1985.
13. D. N. Jansen and R. J. Wieringa. Extending CTL with actions and real-time. In *Proc. International Conference on Temporal Logic 2000*, 2000.
14. F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques*. Prentice Hall, 2000.
15. S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, New York, New York, 1984.
16. OMG. *Unified Modeling Language version 1.3*. OMG, July 1999.
17. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
18. D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F. N. Afrati and P. Kolaitis, editors, *6th International Conference on Database Theory (ICDT)*, LNCS 1186. Springer, 1997.
19. Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. `http://www.wfmc.org`.