

A Formal Semantics for UML Activity Diagrams – Formalising Workflow Models

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

In this report we define a formal execution semantics for UML activity diagrams that is appropriate for workflow modelling. Our workflow models express software requirements and therefore assume a perfect implementation. In our semantics, 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 with multiple state instances. We first give an informal description of our semantics and then formalise this in terms of labelled transition systems. We compare our semantics with other semantics for UML activity diagrams and workflow modelling by analysing the different choices made in those semantics.

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 (a.k.a. workflow specification) is the definition of a workflow. An instance of a workflow is called a *case*. In a case, *work items* are passed and manipulated. Examples of cases are an insurance claim handling instance and a production order handling instance. Examples of work items are claim forms and damage reports. The definition, creation, and management of workflow instances is done by a workflow management system (WFMS), on the basis of workflow models.

In general, two important dimensions of workflows are the process-logic dimension and the resource dimension [2, 20]. The process-logic dimension concerns the ordering of tasks (or activities) in time (what has to be done). The resource dimension concerns the organisational structure (who has to do it). In this report, we focus on modelling the process-logic dimension of workflows. When we use the term workflow model, we refer to a model that describes the process-logic dimension.

Any modelling language should satisfy various requirements. First, it should be expressive enough to model all desired constructs. We look at desired workflow modelling constructs in Section 2. Second, to act as a communication medium, it should be a standard

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

language. Third, to be easily readable, it should be graphical. Fourth, to see whether a model satisfies a certain set of requirements, it should be possible to analyse that model. We distinguish between functional requirements (what should be done) and quality requirements (how well it should be done). Fifth, in order to validate a model, we think it should be possible to execute the model. Then a user can test by executing the model whether the model is valid. The last two points imply that the modelling language must have a formal execution semantics.

In this report, we use UML activity diagrams to model the process-logic dimension of workflows. The UML [23] is a graphical notation for modelling object-oriented systems, proposed as industry standard by the OMG. An activity diagram is a diagram to specify the dynamics of a system. It has constructs to express sequence, choice and parallelism. Its syntax is inspired by both Petri nets [24], statecharts [15] and flowcharts. UML activity diagrams are expressive enough to model all the constructs needed in workflow models (this is argued in Section 4). Unfortunately, the activity diagram semantics defined by the OMG [23] is informal and ambiguous. Moreover, although activity diagrams are intended for workflow modelling, the OMG semantics of activity diagrams is more suitable for software models (this is argued in Section 9).

In this report, we therefore define a formal execution semantics for UML activity diagrams 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 [8] and to use model checking tools (e.g. Kronos [29]) for the analysis of functional requirements.

A secondary goal of this report is a comparison with other formal modelling techniques for workflows, like Petri nets [2] and statecharts [27]. 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. In Section 9 we discuss extensively the similarities and differences between our activity diagram semantics and the Petri net semantics.

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.

The remainder of this report is structured as follows. First we discuss workflow concepts in Section 2. Next, in Section 3 we give an informal semantics of activity diagrams in terms of the domain of workflow. In Section 4 we define the syntax of an activity diagram, and define and discuss constraints on the syntax. In Section 5, we define an activity hypergraph as the underlying syntactic model of a collection of activity diagrams. We also define constraints on activity hypergraphs, and discuss how an activity diagram maps into an activity hypergraph. Then we define specifications of activities. In Section 6 we present a semantics that defines how to execute an activity hypergraph. In Section 7 we present a semantics of an activity hypergraph (based upon Section 6) that can be used to analyse an activity hypergraph by means of model checking. In Section 8 we add dynamic concurrency for activities (dynamic concurrency activities are dynamically instantiated multiple times) to our syntax and semantics. In Section 9 we discuss other formalisations of activity diagrams and other formal languages for workflow modelling. We end with a discussion and conclusions. Formulas are written in the Z notation [26].

2 Workflow Concepts

Workflow modelling. In the previous section, we already explained some concepts of workflows. We elaborate on this. The following exposition is based on literature (a.o. [2, 20]) and several case studies that we did.

Activities are done by *actors*. 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 work items and other data relevant for the case. The case may be distributed over several actors. Each distributed part of the case has a *local state*. There are three kinds of local state.

- 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.

For the remainder of this report, we do not consider queue states anymore, since a queue state can be considered as a wait state, where the event that has to be waited for is that an actor becomes available.

We allow multiple instances of states to be active at the same time. For example, the activity diagram in Fig. 1 shows two parallel activities Produce partial order and Fill partial order that each trigger an instance of Send partial shipment. The result is that two instances of Send partial shipment may be active at the same time. 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.

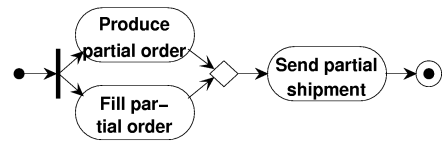


Figure 1: Activity diagram that cannot be translated into a statechart

Actors are people or machines. Actors are grouped according to roles. A *role* describes certain characteristics of actors. A role can refer to skills, responsibility, or authority for people, and it can refer to computing capabilities for machines [20, 28]. Roles link actors and activities. The modelling of actors and roles, and the connection with workflow models falls outside the scope of this report.

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 WFMC [28] specifies four possible ordering relationships between activities: *sequence*, *choice*, *parallelism* and *iteration*. And, to facilitate readability and re-use of workflow definitions, an ordered set of activities can be grouped into one *compound activity*. A compound activity can be used in other workflow definitions. A non-compound activity is called an *atomic activity*.

Example. In the remainder of this report, we use the workflow of a small production company as running example (adapted from [28]). First, an order is received. Next, the departments Production and Finance are put to work. Finance checks whether the customer's account limit is not exceeded by accepting the order. If Finance rejects the order,

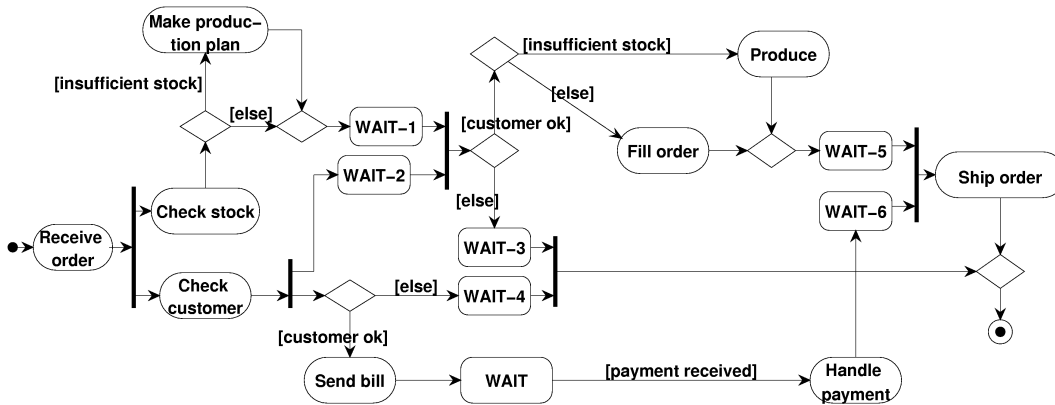


Figure 2: Example activity diagram

the whole workflow stops. Otherwise, Finance sends a bill to the customer and waits until the customer pays. Production checks whether the desired product is still in stock. If not, a production plan must be made to produce the product. If according to Finance the order may be accepted, the product is either produced or taken from stock. If both Production and Finance have finished, the product is shipped to the customer.

Figure 2 shows the activity diagram of this example. Ovals represent activity states, rounded rectangles represent wait states, and arrows represent state transitions. Section 4 explains the details of the notation.

System structure. In Figure 3 we show the system structure we assume in this report. A workflow system (WFS), which is a WFMS instantiated with one or more workflow models, connects a database and several applications that are used by actors to do work for the cases. We abstract away from the communication and user interface domain.

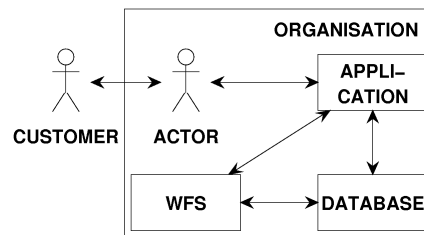


Figure 3: System structure

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 work item by editing it with a word processor. The transitions between the states (active or waiting), on the other hand, are performed 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.

For the remainder of this report, we assume the WFS controls a single case. (Since we do not consider case management, cases are independent from each other. Hence, a generalisation to a WFS that controls multiple cases is straightforward.)

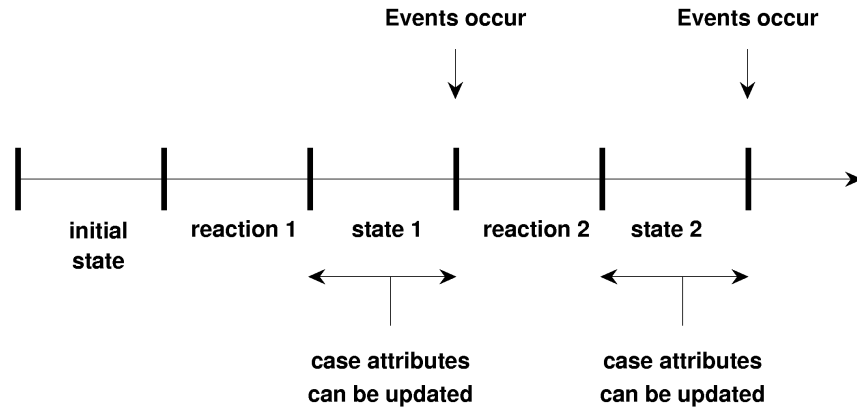


Figure 4: Structure of a run. States can have different durations. Events are reacted to as soon as they occur.

3 Semantics for Workflow Models

First, we identify requirements on semantics for workflow model. Then we explain our execution structure. Finally, we discuss how activities are specified.

3.1 Semantic requirements.

An activity diagram prescribes how a WFS behaves. So, an activity diagram is a requirements specification of the WFS. 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 [17] 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 its environment. The most important characteristic of an reactive system is its continuous interaction with its environment. Often, a reactive system operates in real-time.

Input to a reactive system are the events that occur in its environment. There are three kinds of events:

- 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.
 - 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 (see Figure 4). System reactions are caused by the occurrence of events.

	Semantic requirements	Implication for semantics
1	The environment is modelled	Input events are part of WFS state
2	Perfect technology is assumed	WFS state changes are instantaneous
3	WFS has clock-asynchronous behaviour	WFS reacts immediately with superstep
4	Case attributes are updated by the actors (i.e. the environment) only	No update actions in workflow specification and post-condition is declaratively specified
5	Case attributes subject to data integrity constraints	Activity must have some effect specification and there must exist integrity constraints
6	Activities of actor take time	Activity state is part of WFS state

Table 1: Semantic requirements and implications

In order to give a precise definition of a run (i.e., of its states and reactions) we identified the following semantic requirements, based upon an analysis of the literature and of a number of case studies (see Tab. 1.) Note that these requirements are defined independently from activity diagrams.

First, a reactive system is continuously ready to interact with its environment. Any semantics for a reactive system should therefore somehow incorporate information about the environment. In our semantics, input events, which are discrete changes in the environment, are part of the state of a run.

Second, the goal of our semantics is to support the specification of functional requirements. We distinguish between functional requirements (what should be done) and quality requirements (how well it should be done). Functional requirements should be specified independently of the implementation platform. That is equivalent to making the *perfect technology* assumption: the implementation consists of infinitely many resources that are infinitely fast [21]. Under this assumption, functional requirements will not be influenced by implementation restrictions. Perfect technology implies that in a functional requirements model, 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 part of the environment, they are not assumed to be perfect. They do take time to perform their activities.

Third, a WFS responds as soon as it receives events from its environment (the application or the database). This kind of behaviour is the *clock-asynchronous semantics* of STATEMATE [16]. An alternative is to react at the tick of the clock, called *clock-synchronous semantics* [16]. But this is inappropriate for workflow modelling, since we then must fix a certain time grid. For our semantics, the clock-asynchronous semantics implies that there does not elapse time between the occurrence of events and the subsequent reaction of the WFS.

In the clock-asynchronous semantics, 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, 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* [16]. In Section 6 we define steps and supersteps. The initial state is unstable by

definition.

The latter two requirements together imply that the WFS responds at the same time as events occur. This is called the *perfect synchrony hypothesis* [5]. For our semantics, it implies that the reaction of the WFS occurs at the same moment the WFS receives its new input events. To satisfy perfect synchrony, a WFS implementation must satisfy the following requirement:

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

Fourth, case attributes cannot be updated by the WFS, but they can be updated by actors, that are part of the environment. This implies that case attributes can only be updated in activities, not in transitions of the workflow. Next, it implies that post-conditions must be specified declaratively. An imperatively specified post-condition would mean that the WFS itself computes the outcome of an activity, which is inappropriate for workflow modelling. For our syntax, this implies that transitions between states in the workflow cannot be labelled with operations that have an effect on the case attributes.

Fifth, the semantics should have some transactional properties to prevent a case attribute from being ill-defined by two concurrent accesses. In our semantics, we define data integrity constraints, that forbid certain activity instances to be active simultaneously. In order to specify a data integrity constraint, for each activity some information must be given on its effect. In our semantics, we specify what variables an activity observes (i.e., reads) and what variables it updates (i.e., writes). In our formalisation, we have adopted a definition that is similar to conflict equivalence (that guarantees serialisability of the activities) of database theory [11].

Sixth, activities take time, since they are done by imperfect actors. This implies that activities must be part of the state of a run, not of the reaction. In a reaction, typically activities are started (probably because their predecessors completed).

Finally, we adopt the following properties from STATEMATE [16]:

- More than one event can be input to the WFS at the same time.

The alternative is that only one event can be input to the WFS at the same time. This alternative implies that no two events can occur at the same time, which can only be true if the rate at which events occur in the environment is slower than the rate at which the WFS reads input events (sampling rate). We do not want to impose such a restriction upon the environment and therefore do not make such an assumption.

- The input is a set of events, rather than a queue.

The latter assumption is adopted by the OMG semantics [23], but is more appropriate for a low-level software implementation semantics, in which not all input events can be responded to simultaneously.

- An event is removed from the input immediately after the subsequent step completes.

We adopt this property for the following reason. The intuitive meaning of an event label e on an edge g is that if e occurs while the system is in the source state nodes of g , the system reacts by taking edge g . If event e would never be removed from the input after it has occurred, this intuitive meaning would no longer hold, since e might have already occurred long before the system entered the source state nodes of g .

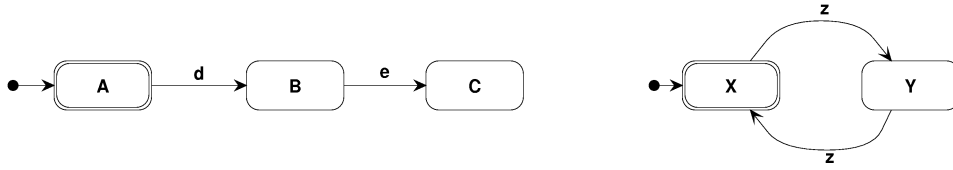


Figure 5: Two example activity diagrams

So events must be removed from the input, but when? There are two possibilities: remove events from the input after the subsequent step completes or remove events from the input after the subsequent superstep completes. If events would be removed after the subsequent superstep instead of after the step, the intuitive meaning sketched above would also no longer hold, since e might have already occurred before the system entered the source state nodes of g . To illustrate this, consider the activity diagram on the left in Fig. 5 and suppose we adopt the remove-event-after-superstep rule. Suppose the system is in state A (indicated by the double lines) and events d and e occur simultaneously. In the subsequent reaction step $[(A,B)]$ is taken and state B is reached. Now the input events d and e stay in the input since the superstep has not yet terminated. Since B is unstable because e is in the input, next step $[(B,C)]$ is taken and state C is reached. Since this state is stable, all input events are removed from the queue. In this example, in state B the system takes edge (B,C) although event e did not occur while the system was in state B. This behaviour is counterintuitive. With the remove-event-after-step rule, on the other hand, the system would take superstep $[(A,B)]$ which is in accordance with the intuitive meaning of the diagram. So in this case, the remove-event-after-step rule is more intuitive.

As a second example, consider the activity diagram on the right in Fig. 5 and suppose we again adopt the remove-event-after-superstep rule. Suppose the system is in state X and event z occurs. Then step $[(X,Y)]$ is taken and state Y is reached. Input event z stays in the input since state Y is not stable. Next, step $[(Y,X)]$ is taken and state X is reached. Again, input event z stays in the input since state X is unstable. Next, step $[(X,Y)]$ is taken and unstable state Y is reached. So, with the remove-event-after-superstep rule the system ends up in an infinite loop! By contrast, with the remove-event-after-step rule the system would take superstep $[(X,Y)]$. This latter behaviour is in accordance with the intuitive meaning of the diagram: upon an occurrence of z the system takes an edge and waits for the next occurrence of z .

We therefore adapt the remove-event-after-step rule.

- If the WFS reacts, i.e. takes a step, it reacts to all events in the input set.

From the previous item, it follows that if the WFS would not react to all events in the input queue, some input events would not be responded to. That is undesirable.

- A step is maximal.

Otherwise some edges that are enabled would not have to be part of the step, so would not have to be taken. Since an event is removed from the input after the subsequent step has been taken, this would mean that some input events would have no effect, although, according to the workflow model, they *should* have an effect (namely the

enabled transition should be taken). In other words, the WFS does not react to these input events. That is why we require that a step be maximal.

3.2 Runs

State of a run. 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 time,
- the current value of the timers. Timers are necessary to generate time-outs.

Remember that 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*. From our discussion on workflow modelling (Section 2), we conclude that each parallel branch has three kinds of possible states. These are:

- an *activity state*, in which the parallel branch waits because some actor is busy performing a certain activity,
- a *wait state*, in which the parallel branch waits for some event to occur,
- a *queue state*, in which the parallel branch waits for some actor to become available.

Routing of the case is represented by the changing of the configuration. A WFS changes configuration during a reaction by changing states, i.e. leaving them, in one or more of the parallel branches that are currently active. A parallel branch can leave a local state for three reasons.

- An activity state is left when the actor finishes doing this activity.
- A wait state is left when an event e occurs and condition c is true, and the workflow model defines an arrow labelled $e[c]$ leaving the corresponding state node.
- A queue state is left when an actor becomes available that can do the next activity.

As stated above: for the remainder of this report, we do not consider queue states anymore; we simply assume that there enough actors available for every activity. A queue states can be modelled as a wait state, where the event that has to be waited for is that the actor becomes available.

Reaction of a run. During a reaction, the state of the case is updated, 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. We precisely define how a WFS reacts in Sections 6 and 7.

3.3 Specifying activities.

In an activity, case attributes can be updated. An activity has a pre and post-condition. The pre-condition specifies when the activity is allowed to start. The post-condition specifies 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 attribute 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. To be precise, what we call non-interference is the isolation property from database theory; see e.g. [11].

Before we define non-interference checks, we must choose the level of aggregation on which we want to detect inconsistencies. There are two possible levels: the level of a database transaction or the level of an activity. An activity usually consists of several database transactions, that are being initiated by actors (cf. Fig. 3). So an activity is at a higher level of aggregation than a database transaction. If we want non-interference on the level of database transactions only, the database can take care of the non-interference. We, however, want to have non-interference on the level of the activity, since we view an activity as atomic (see our definition of activity in Section 2). In general, an activity cannot be modelled as a database transaction, since an activity has a long and a database transaction a short duration. Hence, non-interference cannot be handled by the database. We therefore assume this non-interference check is being done by the WFS (e.g. by means of a transaction processing monitor that is part of the WFS).

Besides having to check for non-interference, having pre and post-conditions has the following consequences for our semantics. When the case is routed, i.e. the WFS reacts and a new configuration is reached, the following things may happen.

- An activity that has to be done next, may have a false precondition.
We postpone execution of the activity until the precondition is true.
- The set of activities that have to be done next, may be inconsistent.
We decide to forbid to start these new activities, i.e., the case cannot be routed in this direction.
- Some of the activities that have to be done next may be inconsistent with the activities that are still being done.
We decide to wait until the conflicting currently executing activities have finished and then start the next set of activities.
- A decision may have to be made on basis of the values of some case attributes that are currently being updated.
We decide to wait until all relevant case attributes are not being updated anymore.

In Section 5 we define precisely how activities are specified, and we define constraints on the syntax such that if these constraints are satisfied, a case is safely routed. In Section 6 we formulate constraints for reactions such that every new configuration is non-interfering.

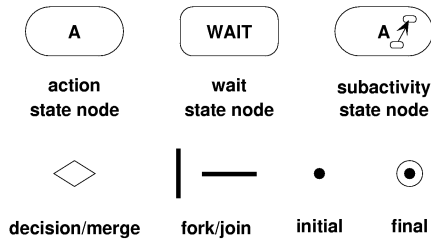


Figure 6: UML activity diagram constructs

4 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 in fact a hypergraph, rather than a graph. We call the underlying hypergraph an *activity hypergraph*. In the next section, we define activity hypergraphs.

This section is organised as follows. Based upon our exposition of workflow concepts in the previous sections, we first choose the UML constructs that are necessary to model workflows. We then motivate why we left out the other UML constructs. Next, we define constraints on the syntax of UML activity diagrams. The definitions we use are more simple than the ones OMG uses [23].

UML constructs used. Figure 6 shows the UML activity diagram constructs we will use. We now discuss these constructs. We use the following UML state constructs for activity diagrams. We use an action state node, to denote an activity state, and a wait state node, to denote a wait state. Next, we use 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. We require the transitive closure of this hierarchy relation between activity diagrams to be acyclic. Besides these state constructs, we use pseudo state nodes to indicate choice (decision state node), merging of different choice branches (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 [3]. State nodes (including pseudo state nodes) are linked by directed edges (expressing sequence) that may be labelled. A label has the form $e[g]/S$ where e is an event expression and g a guard expression and S the set of events that are generated if e occurs and g is true. Empty event `NULL` and guard `true` are not shown on an edge. Below, we will define restrictions on the occurrence of labels. Special event labels are `when(tex)` and `after(tex)`, denoting an absolute and a relative temporal event respectively, where global clock gc measures the current time and tex is an integer expression, counting time-units of the global clock and the local clocks. Periodic events are events that are not specified at a single point in time, but at a sequence of points, for example every day or every week. These events are modelled with the `when(cond) each period` construct.

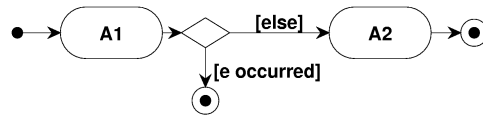


Figure 7: Specifying an interrupt

UML constructs removed.

- We do not label edges with action expressions that have an effect on the case attributes. Updates on activities are performed by actors in activities, not by the WFS in the transitions of a workflow. The only actions we therefore allow are send actions, in which events are generated.
- 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. First, does an object flow state represent the state of a case? If it does, then what happens if the object flow state is forked? Do we have multiple copies of one case, or do we have multiple cases? Second, what happens if an object flow state is input to an activity that may be instantiated multiple times? Is it input to one of the instantiations, or to all of them? If the latter is the case, how is consistency of the object ensured? Third, if an activity has an object flow state as its output, does this mean that the object flow state is created or updated? Fourth, what does it mean when two object flows are merged into one object flow state? How do we construct a consistent case? Fifth, what if no object flow state node is shown in an activity diagram. Does that mean it does not exist, or simply that it is not shown, but does exist?

For the moment, we decide to omit object flow states (and thus object flows) from our syntax, and thus from our semantics. Object flow states seem to have no added value above the use of a database (only one vendor of workflow management systems supports object flow states [20]) and they considerably complicate the semantics. Instead, we represent the case attributes by the local variables of the activity diagram and assume these attributes are stored in a database (cf. Fig. 3).

Constraints on activity diagrams. We now formulate constraints on the syntax. These constraints facilitate the mapping of an activity diagram to an activity hypergraph. A constraint is labelled **UML** if it is defined in UML 1.3 [23]. The other constraints are defined by us.

1. **UML** No edge that leaves an action state node is labelled with an event.

An action state node represents the execution of an atomic activity. An atomic activity cannot be interrupted.

Note: this a soft constraint that may be violated. In our semantics, however, the edge can only be taken if the activity has terminated, so the atomicity of the activity is not violated.

2. No edge that leaves a subactivity state node, is labelled by an event.

An edge with an event label specifies an interrupt: the edge is taken when the event occurs. For compound activities, this would imply that an atomic subactivity can always be interrupted, which is not true. A compound activity can only be interrupted in between two atomic subactivities. But this can be shown in the activity diagram that specifies the subactivity. For example, if we want event e to interrupt compound activity A , and A is decomposed into atomic subactivities $A_1;A_2$ then we must introduce a decision between A_1 and A_2 to check whether the interrupt occurred (see Figure 7).

3. **UML** Edges leaving a fork state node are not labelled by guards.

This rule rules out possible deadlocks, since if one of the guards is false, a subsequent join may get stuck.

4. From every (pseudo) state node there goes at most one edge into a join state node.

Otherwise there would be a deadlock, since only one edge can be taken.

5. **UML** Edges leaving a decision state node are labelled only with guard expressions. Moreover, the disjunction of guard expressions on the edges leaving a decision state node must evaluate to **true**. We choose to check this at compile time by requiring that this disjunction must be a tautology.

The latter constraint avoids that the execution gets stuck, if every edge that leaves the decision state node has a guard expressions that evaluates to **false**. Ideally, the choice is deterministic, but it can be nondeterministic (if two or more edges leaving the decision state node, have a guard expression that evaluates to **true**). An edge, leaving the decision state node, may be labelled with guard expression **else**, which is a shorthand for the negation of the conjunction of all other (non-else) guard expressions that label edges leaving the decision state node. For example, if a decision state node has two outgoing edges, and if the guard expression on one of the edges is $[x \leq 10]$, the guard expression **else** on the other edge is a shorthand for $[\text{not } (x \leq 10)]$.

6. Every (pseudo) state node should have at least one incoming and outgoing edge, except the initial state node (only has outgoing edges) and final state node (only has incoming edges).

This constraint ensures that every pseudo state node can be eliminated. For an ordinary state node s , the constraint merely states that s should be connected to some other state node (an unconnected state node makes no sense, since it is unreachable).

7. **UML** For every fork state node, there is only one incoming edge. For every join state node, there is only one outgoing edge.

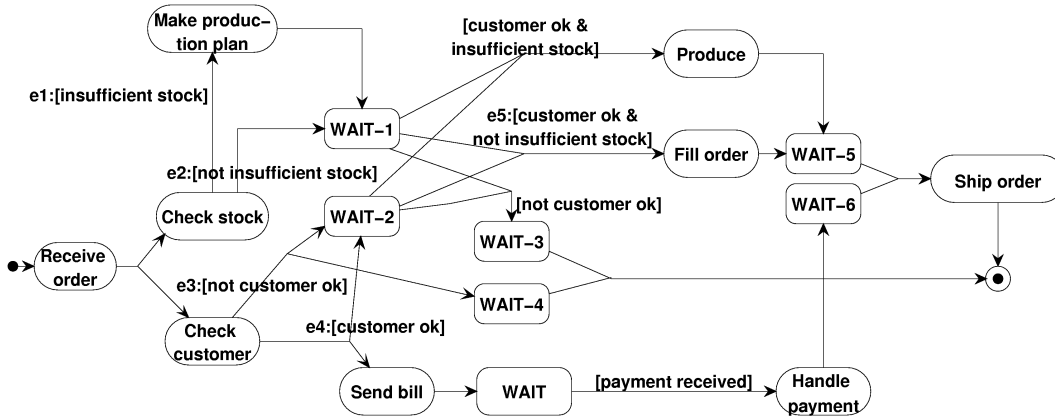


Figure 8: Activity hypergraph of our running example

These two constraints enhance readability of the activity diagram. They do not limit its expressiveness.

8. For every state node s and every join state node j , if there is an edge from s to j , then s should be a wait state node.

A join state node represent synchronisation of two or more parallel branches. Since not every branch has to reach the join state node at the same time, there must be a wait state node for every branch.

9. **UML** An edge that leaves the initial state node is not labelled with an event.

The initial state node is interpreted as an unstable state: it is immediately left when the case starts executing the activity diagram. If it were allowed to label an edge with an event expression e , the initial state node might not be left, since e might not occur.

10. Every activity diagram should have exactly one final state node.

If an activity diagram A is used as specification of a subactivity state node, the intended meaning is that the subactivity state node will be left if the whole activity diagram A has completed executing. The above constraint more or less enforces this, although there are extreme situations in which this constraint is not strong enough (e.g., a fork followed by a merge). A stronger constraint, however, will rule out activity diagrams having no undesired behaviour.

5 Activity Hypergraphs

In this section, we define the hypergraph structure that underlies an activity diagram. This structure we call an activity hypergraph. Figure 8 shows the activity hypergraph corresponding to Figure 2. Below, we discuss the syntax of activity hypergraphs. In the next section, we define their semantics.

Syntax of activity hypergraphs. An activity hypergraph is a rooted directed hypergraph. We assume given a set *Activities* of activities. An *activity hypergraph* is a quintuple $(Nodes, Edges, Events, Guards, LVar)$ where:

- $Nodes = AS \cup WS \cup \{initial, final\}$ is the set of state nodes,
- $Edges \subseteq \mathbb{P} Nodes \times Events \times Guards \times \mathbb{P} Events \times \mathbb{P} Nodes$ 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* 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.

Given $e = (N, ev, g, ac, N') \in Edges$, we define $source(e) \stackrel{\text{df}}{=} N$, $event(e) \stackrel{\text{df}}{=} ev$, $guard(e) \stackrel{\text{df}}{=} g$, $action(e) \stackrel{\text{df}}{=} ac$, and $target(e) \stackrel{\text{df}}{=} N'$.

State nodes *initial* and *final* denote the initial and final state node, respectively. Besides these special state nodes, an activity hypergraph has action state nodes *AS* and wait state nodes *WS*. Every action state node has an associated activity it controls, denoted by the function $control : AS \rightarrow Activities$. The execution of the activities falls outside the scope of the activity hypergraph, 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.

We assume a set $BE(LVar)$ of boolean expressions on set *LVar*. We assume a set of clocks $Clocks \stackrel{\text{df}}{=} \{gc\} \cup LocalClocks$ where $LocalClocks \stackrel{\text{df}}{=} \{lc(e)(n) \mid e \in Edges \wedge n \in \mathbb{N}\}$, consisting of the global clock *gc* that measures the current time and for every edge *e* a local clock $lc(e)(n)$ where $n \in \mathbb{N}$ that we use for specifying time-outs. The global clock is never reset, whereas at least one local clock $lc(e)(n)$ is reset to zero every time the sources of *e* are entered (see Section 7 for details). We assume a set $BE(Clocks)$ of boolean expressions on clocks. Every basic clock constraint $\phi \in BE(Clocks)$ has the form $c = \text{exp}$ where $c \in Clocks$ and $\text{exp} \in \mathbb{N}$. This definition of $BE(Clocks)$ is sufficient for providing a semantics for the UML **when** and **after** constructs.

Set *Guards* is constructed as the union of $BE(LVar)$ and $BE(Clocks)$ and the set of expressions that is obtained by conjoining (\wedge) elements of the sets $BE(LVar)$ and $BE(Clocks)$. We require that if $lc(e)(n) = \text{exp}$ is part of the guard expression of edge e' , that $e = e'$.

Syntactic constraints on activity hypergraphs.

1. Every edge that leaves an action state node is labelled with empty event NULL.

$$e \in Edges \wedge source(e) \cap AS \neq \emptyset \Rightarrow event(e) = \text{NULL}$$

2. For every edge *e* that has action state node *a* as source, *a* is the only source of *e*. This implies that for every edge with multiple sources, none of its sources is an action state node.

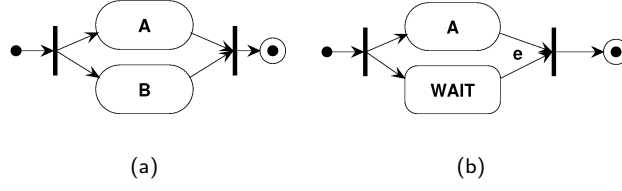


Figure 9: Two forbidden examples of completion before join

$$e \in Edges \wedge a \in source(e) \cap AS \Rightarrow source(e) = \{a\}$$

This constraint is similar in spirit to constraint 8 for activity diagrams. For example, we disallow Fig. 9(a) and 9(b). It is impossible to give a semantics to the join that in Fig. 9(a) does not require A and B to complete simultaneously, and that in Fig. 9(b) does not require A to complete exactly when e occurs. Of course, we can easily treat (a) and (b) as shorthand for diagrams in which wait state nodes have been inserted before the join.

3. The disjunction of guards on the edges leaving an action state node must be a tautology.

$$a \in AS \Rightarrow \bigvee \{guard(e) \mid \exists e \in Edges \bullet a \in source(e)\}$$

This constraint is similar to constraint 5 for activity diagrams.

4. The initial state node may only occur in the source of an edge. Moreover, if it is source of an edge, it is the only source of that edge.

$$initial \in source(e) \cup target(e) \Rightarrow \\ initial \notin target(e) \wedge source(e) = \{initial\}$$

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.

$$final \in source(e) \cup target(e) \Rightarrow \\ final \notin source(e) \wedge target(e) = \{final\}$$

This constraint is similar to part of constraint 6 on activity diagrams.

5. The edges leaving the initial state node must have no events and the disjunction of their guard expressions must be a tautology.

$$source(e) = \{initial\} \Rightarrow event(e) = \text{NULL} \\ \bigvee \{guard(e) \mid e \in Edges \wedge source(e) = \{initial\}\}$$

This constraint ensures that the initial state is unstable. The first part of the constraint is similar to constraint 9 on activity diagrams.

These constraints are not restrictive enough to rule out all undesired activity hypergraphs. But more restrictive constraints would rule out several activity hypergraphs that have no undesired behaviour. To find undesired properties in activity hypergraphs, we will use model checking.

Mapping an activity diagram to an activity hypergraph. The mapping consists of the following steps.

1. Eliminate hierarchy
2. Eliminate pseudo state nodes
3. Rewrite temporal events

1. We eliminate hierarchy as follows. In the UML, subactivity state nodes may appear. As stated above, we assume the behaviour of a subactivity state node is modelled by another activity diagram, that we call subactivity diagram. We require that the transitive closure of the hierarchy relation is acyclic. We can then eliminate all subactivity state nodes from the activity diagram by substituting for every subactivity state node the appropriate activity diagram (a kind of macro-expansion). Note that we regard the initial and final state nodes of these subactivity diagrams as pseudo state nodes. These pseudo state node can be eliminated by conjoining every compound transition (see below) to (from) the subactivity state node with every compound transition that leaves (enters) the initial (final) state node of the subactivity diagram.

2. We eliminate pseudo state nodes as follows. Following Harel and Naamad [16], we use the concept of a compound transition to map edges in the activity diagram to hyperedges in the activity hypergraph. Every compound transition in the activity diagram is a hyperedge in the activity hypergraph. A compound transition is a maximal chain of edges, linked by pseudo state nodes. There are two types of pseudo state nodes: AND and OR. For the AND pseudo states (fork and join), every edge to or from the pseudo state node is part of the same compound transition. For every OR state node (decision and merge) exactly one of the incoming and one of the outgoing edges is part of the compound transition.

3. Every hyperedge e labelled with $\text{when}(gc = \text{exp})$ is replaced by a hyperedge e' that has clock constraint $gc = \text{exp}$ which is conjoined with e 's guard expression. Every hyperedge e that is labelled with an $\text{after}(\text{exp})$ constraint is replaced by infinitely many hyperedges $e(n)$ where $n \in \mathbb{N}$, each edge $e(n)$ labelled with clock constraint $lc(e)(n) = \text{exp}$ that is conjoined with e 's original guard expression. In Section 7 we give an example that explains the semantics of after and that motivates why infinitely many hyperedges are needed.

The mapping of the $\text{when}(\text{cond})$ *each period* construct is more involved. To keep this presentation simple, we assume that every periodic event labels an edge whose source is a single wait state node and whose target is a single action state node. A generalisation to hyperedges (multiple source and targets) is straightforward, provided there does not exist an instantaneous superstep from target to source: that would result in divergence. ¹Every wait state node w of an activity diagram that is source of an edge labelled with a periodic event $\text{when}(\text{cond})$ *each period*, we map into two wait state nodes w_1, w_2 of the activity hypergraph. One wait state node, say w_1 , represents that the periodic event must happen this period;

¹The now following translation was suggested by D'Argenio [7].

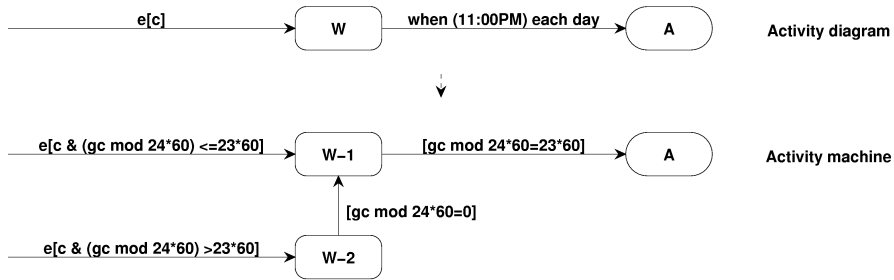


Figure 10: Example translation of periodic events

the other node, say w_2 , represents that we were too late for this period and therefore have to wait for the next period for the periodic event to occur. So, wait state node w_1 is entered if the periodic event did not yet occur this period. Wait state node w_2 is entered otherwise. For example, if the period event is specified by `when (11:00PM) each day`, and the current time is 3PM, node w_1 is entered because 3PM is before 11PM and before 12PM (the end of the period), and w_2 is entered otherwise. An hyperedge that enters w is mapped into two hyperedges, one entering w_1 with an extra condition label $[(gc \bmod period) \leq texp]$, the other one entering w_2 with an extra condition label $[(gc \bmod period) > texp]$. We define a new edge from w_2 to w_1 with label $[gc \bmod period = 0]$. The edge is taken if the new period has begun. State w_1 is left if the condition $(gc \bmod period) = texp$ becomes true, which means that the periodic event occurs. An example translation is shown in Figure 10.

Finally, every ordinary state node (including the one introduced by the translation periodic events) in the activity diagram maps into a unique state node in the activity hypergraph.

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. Note however that it is possible to have $Obs(a) = Upd(b)$, if $a \neq b$.

Two activities are in *conflict*, 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 [11].)

$$\begin{aligned}
 A \nabla B &\Leftrightarrow (Obs(A) \cup Upd(A)) \cap Upd(B) \neq \emptyset \\
 &\vee (Obs(B) \cup Upd(B)) \cap Upd(A) \neq \emptyset
 \end{aligned}$$

Note that this particularly implies that we allow autoconcurrency (two instances of the same activity that are active at the same time) only if the activity does not update variables.

Furthermore, we define for every activity a a pre and post-condition, $pre(a)$, $post(a)$. 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 $\sigma : LVar \rightarrow \mathcal{D}$ 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 $\Sigma_p(LVar)$. 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. For type integer, the set of legal expressions would be all boolean expressions, composed of operators $<, \leq, >, \geq, =, \neq$ applied to integer variables or integer values. Since we have defined no formal syntax for pre and post-conditions, we do not provide a formal semantics for the satisfaction relation \models , but in our semantics we simply assume a formal syntax and semantics for pre and post-conditions has been chosen.

Function $effect : \Sigma_p(LVar) \times AS \rightarrow \mathbb{P} \Sigma(LVar)$ 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 $effect(\sigma, a) = \{\sigma' \mid \sigma' \models post(a) \wedge \forall v \in Obs(a) \bullet \sigma(v) = \sigma'(v)\}$.

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 $effect(\sigma, a)$ is a possible effect of activity a , the combined effect of activities in A is $\uplus_{a \in A} effect(\sigma, a)$, where \in is bag membership and \uplus is bag union. 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 $effect(\sigma, A)$.

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

Constraints. On page 9, we made several decisions to prevent illegal routing of a case. The following constraints on activity hypergraphs prevent a case from being illegally routed.

1. For every action state node that controls an activity with a precondition, there should be a preceding wait state node. Our semantics will take care that that the system stays in the wait state node as long as the precondition is **false**: if the pre-condition is **true**, our semantics will ensure that the wait state node is skipped: the action state node is immediately entered and the activity is started. Next, our semantics ensures that the precondition can only be evaluated iff all updated and observed variables it refers too, are not being updated anymore.
2. 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. Then a routing decision can be made safely.

These two constraints are illustrated by means of an activity diagram in Fig. 11. Note that these constraints apply to activity hypergraphs. As syntactic sugar, in an activity diagram we omit the wait state nodes preceding and following an action state node that controls an activity with a pre and post-condition.

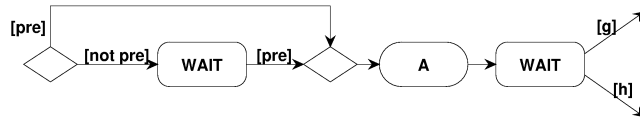


Figure 11: Modelling pre and post-conditions in activity diagrams

6 Execution semantics of Activity Hypergraphs

In this section, we define the formal execution semantics of an activity hypergraph. We first give some more information on reactive systems. We then proceed by defining, how a system reacts, given a system state and a set of events that occurred.

For analysis purposes, the execution semantics needs to be augmented with information on how the environment behaves. In the next section, we will present a semantics for analysis purposes, which is based on the execution semantics of this section.

States. At each point in time, a system state consists of

- the current configuration C ,
- the current set I of inputs,
- the current value of every local variable $v \in LVar$,
- the current value of the global clock gc ,
- the current value of the running local clocks $On \subseteq LocalClocks$.

Formally, the system state is a valuation σ :

$$\sigma : \{C, I\} \cup LVar \cup \{gc\} \cup On \rightarrow \mathcal{D}$$

In Section 7 we will precisely define how set On changes if a step is taken.

Configurations. The configuration is a multiset of state nodes $Nodes \rightarrow \mathbb{N}$ of the activity hypergraph. (A multiset rather than a set since multiple copies of a state node might be active at the same time.)

A configuration is non-interfering, written $non\text{-}interfering(C)$, iff it does not contain non-interfering action state nodes:

$$non\text{-}interfering(C) \stackrel{\text{def}}{=} \forall a, a' \in C \bullet \neg (a \zeta a')$$

In the sequel, the only configurations we allow are non-interfering ones.

Input. We define input I to be a tuple $(Ev, \sigma_p^{LV}, T, \sigma_p^t) \in Events \times \Sigma_p(LVar) \times (AS \rightarrow \mathbb{N}) \times \Sigma_p(Timers)$, 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: $\{NULL\} \subseteq Ev$. Partial valuation σ_p^{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: $T \sqsubseteq (AS \triangleleft C)$, where \sqsubseteq denotes the sub-multiset relation and $X \triangleleft Y$ denotes restriction of relation Y to the domain set X . Finally, partial valuation σ_p^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 σ_p^t is running: $\sigma_p^t \subseteq (On \triangleleft \sigma_p^t)$.

Remember we forbade the routing of the case when the routing decision had to be made on basis of a variable that was currently being updated in an activity. We decided to wait until the variable was not updated anymore and to route the case then. One of the possible side effects of a terminated action state node is that some other part of the case now can be routed, because some relevant variable is not updated anymore. In the superstep that is the response to the completion event, this routing is done.

Computing steps. If the environment of the system changes, the system reacts immediately by taking a step. A step is a maximal, consistent sub-multiset of the multiset of enabled edges. In addition, the new configuration must be non-interfering. We explain and define these concepts below. Our definitions extend and generalise both the STATEMATE semantics [16] and our semantics for UML statecharts [12] from sets of states (edges) to multisets of states (edges).

Before we define the multiset of enabled edges $En(C, Ev, T)$, 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. possibly ill-defined value of a variable. 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 σ_p^t because these are contained in σ . In formulas:

$$\begin{aligned}
En(C, Ev, T) &\stackrel{\text{df}}{=} \{e \mapsto n \mid ms(source(e)) \sqsubseteq C' \wedge event(e) \in Ev \\
&\quad \wedge eval(\sigma, guard(e)) \wedge \sigma \models guard(e) \\
&\quad \wedge n = \min(\{C' \# s \mid s \in source(e)\}) \} \\
&\quad \text{where } C' = (((Nodes - AS) \triangleleft C) \uplus T) \\
eval(\sigma, g) &\stackrel{\text{df}}{\Leftrightarrow} \forall a \in AS \triangleleft (C \uplus T) \bullet var(g) \cap Upd(a) = \emptyset
\end{aligned}$$

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

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

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

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

$$nextconfig(C, E) \stackrel{\text{df}}{=} C \uplus \uplus_{e \in E} ms(source(e)) \uplus \uplus_{e \in E} ms(target(e))$$

We require that taking a step E leads to a non-interfering new configuration.

$$non-interfering(nextconfig(C, E))$$

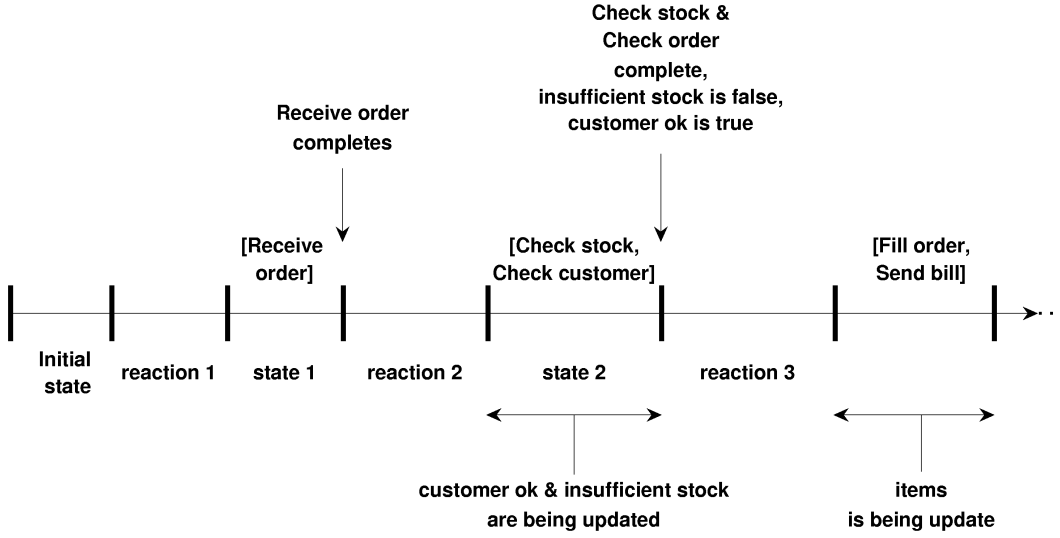


Figure 12: Run of our running example. In each state, the set of activities currently executing is shown.

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

$$\begin{aligned} \text{maximal}(C, Ev, T) \stackrel{\text{df}}{\iff} & \forall e \in \text{En}(C, Ev, T) \mid e \notin E \bullet \neg \text{consistent}(C, E \uplus \llbracket e \rrbracket) \\ & \vee \neg \text{non-interfering}(\text{nextconfig}(C, E)) \end{aligned}$$

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.

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

Note that for a given configuration C and set of input events I there may more than one set of enabled edges that satisfies isStep . So, the system can be nondeterministic.

Algorithm. The algorithms in Figure 13, 14 and 15 serve as an informal explanation of the semantics. In Section 7 we will formally define the algorithms in Figures 13 and 14, but will stick to the isStep equivalent of the algorithm in Figure 15.

Example. Consider the example activity diagram in Fig. 2 and its underlying activity hypergraph in Fig 8. Let activity Check stock set the boolean variable `insufficient stock`, Check customer the boolean variable `customer ok`, and Fill order the boolean variable `items`. We assume there are no conflicting action state nodes, and no pre and post-conditions. Figure 12 shows part of a run of our example. In this example run, reaction 3 is most interesting. In our example run, in configuration `[Check stock, Check customer]`, both activities Check stock

```
Initialise the configuration;
Initialise the set of input events;
Initialise the set of timers;
Execute a superstep (see Figure 14);
While true do
    Wait for input events from the environment;
    Receive input events;
    React to input events by executing a superstep (see Figure 14);
od
```

Figure 13: Execution algorithm

```
Repeat
    Compute a new step (see Figure 15);
    Compute the new configuration;
    Update the configuration;
    Reset the input;
    Reset all relevant timers;
Until there is no enabled edge
```

Figure 14: Algorithm for executing a superstep

```
Compute the bag of enabled edges from the configuration and the input;
Compute all consistent subbags of enabled edges;
Remove from these consistent bags those that have conflicting nextconfigurations;
Pick one of the largest (wrt bag containment) bags of the remaining bags;
```

Figure 15: Algorithm to compute a step

Completing activity	insufficient stock			
	true		false	
	customer ok		customer ok	
	true	false	true	false
Check stock	[Check customer, Make production plan]	[Check customer, Make production plan]	[Check customer, Wait-1]	[Check customer, Wait-1]
Check customer	[Check stock, Wait-2, Send bill]	[Check stock, Wait-2, Wait-4]	[Check stock, Wait-2, Send bill]	[Check stock, Wait-2, Wait-4]
Check stock & Check customer	[Make production plan, Wait-2, Send bill]	[Make production plan, Wait-2, Wait-4]	[Fill order, Send bill]	[final]

Table 2: Next configurations

and Check customer complete at the same time, guard insufficient stock is false, and guard customer ok is true. The reaction starts by taking step [e2,e3,e4] and entering configuration [WAIT-1, WAIT-2, Send bill]. This state is unstable, since the hyperedge e5, outgoing Wait-1 and Wait-2, is enabled. Therefore, step [e5] is taken and configuration [Fill order, Send Bill] is reached. Since this configuration is stable, the system stops reacting.

In order to give the reader an impression of all the different execution possibilities in configuration [Check stock, Check customer], we have listed in Table 2 all the relevant inputs that may occur, and all the configurations that are the result of the reactions to these inputs. (The superstep that was taken in reaction to a certain input can be derived from the reached configuration.)

Note that in the example, the only edges that might be inconsistent represent different branches of a decision. Furthermore, every decision in the example is deterministic. Hence, no two edges that might be inconsistent are enabled at the same time. Therefore, for each configuration and set of input events, the calculated step is unique. Furthermore, since we have no autoconcurrency (i.e. in each configuration a state node is not contained more than once), both configurations and steps are sets rather than multisets.

7 LTS Semantics of Activity Hypergraphs

In this section we define a semantics for activity hypergraphs, based on the execution semantics of the previous section, that is suited for analysis purposes. Besides defining precisely how the system behaves (in a step), we precisely define how the environment behaves. This semantics is an adaptation of the semantics we previously defined for UML statecharts [12].

We define this semantics in terms of a labelled transition system (LTS). An LTS is a set of states connected by a directed labelled transition relation. Two states s, s' are related by the transition relation iff the system can change state from s to s' . Below, we will give a formal definition of an LTS.

There are two reasons for choosing an LTS as a representation of the behaviour for analysis purposes. First, a system may have infinitely many runs. In that case analysis of all system behaviour is impossible. An LTS offers a finite representation of a infinite set of runs (just like a finite state machine is a finite representation of an infinite set of traces). Second, it is not possible to refer to choices in runs, since choices are not modelled by runs.

In an LTS, all choices are explicitly modelled.

Besides, there exist property languages for specifying functional requirements on LTSs, and a number of model checking tools that can check whether an LTS satisfies a certain property. The advantage of model checking tools over theorem-proving tools is that model checking is fully automatic, and that if a property is violated, a trace (scenario) that leads to the violation of the property is produced by the model checking tool.

Clocked Labelled Kripke Structure. Before we define our LTS structure, which is a Clocked Labelled Kripke Structure (CLKS), we give a short explanation. As stated above, an LTS consists of states and transitions. Just like in the previous section, a (system) state consists of the values of the current input, the current configuration, and the global clock and the local clocks (timers). Thus, a state in the CLKS is a snapshot of the system. We do not, however, represent explicitly the value change and temporal events, since these are already represented by the valuation of the local variables and the clocks. In the semantics defined below, these events are sensed implicitly by the system. Thus, the input is just a subset of named external events and completion events (the first and third component of the input tuple of the previous section).

The CLKS has two kinds of transitions. The first kind of transition, *time transition*, represents the elapsing of time. As such, it is non-instantaneous. In a time transition, only the clocks are updated. The second kind of transition is a *data transition*, which represents an instantaneous change of the state of the system (instantaneous implies that the clock values are not changed). Below, we will define two data transitions: one representing the reception of new inputs, the other one representing the reaction of the system to these new inputs.

We now proceed with the formal definition. We assume a set of typed variables Var and a typed data domain \mathcal{D} . As before, the set of type-preserving valuations defined on Var is denoted $\Sigma(Var)$. Every state of the Clocked Labelled Kripke Structure defined below is a valuation σ of the variables Var . Valuations are connected by transitions labelled with actions out of a given set Act .

A Clocked Labelled Kripke Structure (CLKS) is a structure $(Var, Act, \rightarrow, ci, \sigma_0)$ with

- Var the set of variables,
- Act the set of actions,
- $\rightarrow \subseteq \Sigma(Var) \times \mathbb{P} Act \times \Sigma(Var)$ the transition relation,
- ci the clock invariant. An assertion that has to hold for clocks during every step. It is only used to raise temporal events,
- $\sigma_0 \in \Sigma(Var)$ the initial valuation.

Set Var consists of

- variable C that represents the current configuration,
- variable Ev that represents the set of named external events,
- variable T that represents the set of terminated action state nodes,

- set $LVar$ of local variables,
- global clock gc that measures the current time,
- set On of running local clocks.

All these variables were already defined in the previous section. We have omitted from Var the I components σ_p^{LV} and σ^c since these are already modelled by $LVar$ and $Clocks$ respectively.

There are various way to define Act . We choose $Act = \mathbb{P}(Events \cup Guards)$ and define a function $act : Edges \rightarrow Act$ as follows:

$$act(E) \stackrel{\text{df}}{=} \bigcup_{e \in E} act_1(e)$$

$$act_1(e) \stackrel{\text{df}}{=} event(e) \cup guard(e)$$

Given an activity hypergraph, its CKS is constructed as follows. First, we specify the clock invariant. For every basic clock constraint $lc(e)(n) = \text{exp}$, we specify a constraint ϕ of the form $lc(e)(n) \in On \Rightarrow lc(e)(n) \leq \text{exp}$. For every edge e with a basic clock constraint $gc = n$, we specify a constraint ψ of the form $source(e) \sqsubseteq C \Rightarrow gc \leq n$. Below, we will see that $lc(e)(n) \in On \Rightarrow source(e) \sqsubseteq C$. But the converse implication does not hold. Clock invariant ci is the conjunction of all constraints ϕ and ψ . We evaluate a clock invariant ci in a valuation σ , and write $\sigma \models ci$ if the clock invariant is true.

Then we specify the transition relation. The transition relation \rightarrow of a CLKS is 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 step is taken because the initial state is by definition unstable. A cycle is structured as follows. First time elapses ($\rightarrow_{\text{timestep}}$). Then an event in the environment occurs: either a named external event occurs, a local variable changes value or some action state nodes terminate, or a temporal event occurs ($\rightarrow_{\text{event}}$). Only local variables that are not being observed or updated by some activity can be changed, otherwise there is a conflict. Note that there is a small difference with the previous section, where we allowed a local variable to change, but to keep its old value. But in that case, the system state cannot become unstable, since there the value of the variable remains the same. When the change occurs, the system reacts immediately ($\rightarrow_{\text{superstep}}$). Note that the relation $\rightarrow_{\text{cycle}}$ is not part of the transition relation \rightarrow of the CLKS!

$$\rightarrow_{\text{cycle}} \stackrel{\text{df}}{=} \rightarrow_{\text{timestep}}^+ \ ; \ \rightarrow_{\text{event}} \ ; \ \xrightarrow{A} \text{superstep}$$

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

$$\sigma \rightarrow_{\text{timestep}} \sigma' \stackrel{\text{df}}{\iff} \exists \Delta \in \mathbb{R} \mid \Delta > 0 \bullet \sigma' = \sigma[\&_{c \in Clocks} c / \sigma(c) + \Delta]$$

$$\text{such that } \forall \delta \in [0, \Delta) \bullet \sigma[\&_{c \in Clocks} c / \sigma(c) + \delta] \models ci$$

Relation $\rightarrow_{\text{event}}$ 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 $\sigma'_p \subset \sigma'$ 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).

This corresponds precisely to the occurrence of named events, value change events and temporal events.

$$\begin{aligned}
\sigma \rightarrow_{event} \sigma' &\stackrel{\text{def}}{\iff} (\forall c \in On \bullet \sigma(c) = \sigma'(c)) \wedge \sigma(C) = \sigma'(C) \\
&\wedge \forall v \in LVar; \forall a \in AS \mid a \in \sigma(C) \uplus \sigma'(T) \bullet \\
&\quad v \in Obs(a) \cup Upd(a) \Rightarrow \sigma(v) = \sigma(v') \\
&\wedge (\sigma'(Ev) \subseteq Events \wedge \{NULL\} \subset \sigma'(Ev) \\
&\quad \vee \exists L \subseteq LVar \mid L \neq \emptyset \bullet \\
&\quad (\forall a \in AS \mid a \in \sigma(C) \bullet L \cap (Obs(a) \cup Upd(a)) = \emptyset) \\
&\quad \vee \exists T_{new} \bullet \llbracket \bullet \rrbracket \neq T_{new} \sqsubseteq (AS \triangleleft (\sigma(C) \uplus \sigma(T))) \wedge \\
&\quad \quad \exists \sigma'_p \in effect(\sigma, T_{new}) \bullet \sigma'_p \subset \sigma' \wedge \\
&\quad \quad \sigma'(T) = \sigma(T) \uplus T_{new} \\
&\quad \vee \nexists \sigma'' \bullet \sigma \rightarrow_{timestep} \sigma'')
\end{aligned}$$

Finally, the system reacts by taking a superstep ($\rightarrow_{superstep}$). A superstep is a chain of steps. All the intermediate states are unstable. The superstep stops executing if a stable state is reached. 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.

$$\begin{aligned}
\frac{A_1 \cup A_2 \rightarrow_{superstep}}{\sigma \xrightarrow{A} \sigma'} &\stackrel{\text{def}}{\iff} (\rightarrow_{unstable} \ ; \ \frac{A_1}{\rightarrow_{step}} \ ; \ \frac{A_2}{\rightarrow_{superstep}}) \cup \rightarrow_{stable} \quad (1) \\
&\stackrel{\text{def}}{\iff} \exists E \mid isStep(E) \bullet \\
&\quad \exists S_1 \subseteq Timers \mid OffTimers(\sigma(C), E, \sigma(On), S_1); \\
&\quad \exists S_2 \subseteq Timers \mid NewTimers(\sigma(C), E, \sigma(On), S_2) \bullet \\
&\quad \sigma' = \sigma[C/nextconfig(\sigma(C), E), Ev/\emptyset, \\
&\quad \quad T/\sigma(T) \uplus (AS \triangleleft \uplus_{e \in E} source(e)), \\
&\quad \quad \&_{s \in S_2} s/0, On/\sigma(On) - S_1 \cup S_2] \\
&\quad \wedge A = act(E) \\
\sigma \rightarrow_{unstable} \sigma' &\stackrel{\text{def}}{\iff} \sigma = \sigma' \wedge En(\sigma(C), \sigma(Ev), \sigma(T)) \neq \emptyset \\
\sigma \rightarrow_{stable} \sigma' &\stackrel{\text{def}}{\iff} \sigma = \sigma' \wedge En(\sigma(C), \sigma(Ev), \sigma(T)) = \emptyset
\end{aligned}$$

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. 6);
- 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 further updated into σ' by computing the next configuration when step E is performed (using the function $nextconfig$ defined in Sect. 6), and resetting the input, removing from T the terminated action state nodes that are left (i.e. the terminated activities responded to), and finally resetting all the new timers in S_2 and 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 $rel(C)$ contains each edge whose source is contained in C .

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

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.

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

Initial configuration. In the initial valuation σ_0 , the configuration only contains one copy of *initial*, there are no input events, and all local clocks are set to zero. The global clock must be given an appropriate value.

Real-time example We illustrate the semantics of the UML *after* construct by means of an example. In Fig. 16 from top to bottom, the activity diagram is shown, the translation of the activity diagram into an activity hypergraph without translation of the *after* constraint, and the translation of the *after* constraint.

Let the configuration $C = [WAIT-1, WAIT-2]$. If event e occurs, step $E = t2$ is taken, and configuration $[WAIT-2, WAIT-3]$ is entered. According to the definition of \rightarrow_{step} , variable on is updated (it was empty) with a new timer t . For this timer t predicate $isNewTimerSet(C, E, \{t\}, on)$ must hold. There are infinitely many singleton sets that satisfy this predicate: in fact, all singleton sets $\{lc(t4)(i)\}$, where $i \in \mathbb{N}$. But only one of these singleton sets is picked! Let us

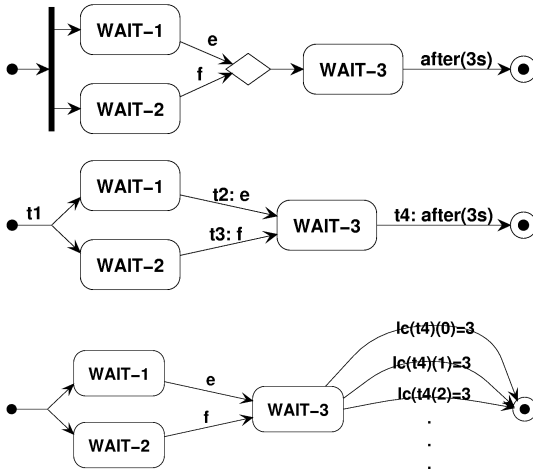


Figure 16: Example mapping of after constraint

for this example assume that it is $lc(t4)(13)$. So variable on is updated with $lc(t4)(13)$ and $lc(t4)(13)$ is started.

Suppose next event f occurs. There are three possibilities: (1) either f happens before 3 seconds have elapsed, (2) or after 3 seconds have elapsed, (3) or simultaneously with the time-out. In each of these cases a new timer must be started, but not all timers can be picked in each case.

(1) Since $lc(t4)(13)$ is already busy, we must pick another timer. Again, we have infinitely many possible timers: we pick, say, $lc(t4)(23)$.

(2) Now $lc(t4)(13)$ is available again, since the timeout already occurred. So we can pick an arbitrary timer.

(3) Since $lc(t4)(13)$ is still busy, we must pick another timer. We can pick for example $lc(t4)(23)$.

Next, note that in this example we do not need infinitely many hyperedges for $t4$, since at most four timers are needed simultaneously. This is because state $WAIT-3$ can have two instances at most. In general, if an hyperedge e is at most n times relevant for an arbitrary configuration, then $2 \times n$ timers are needed. Unfortunately, such a bound on the number of relevant hyperedges does not have to exist. But if such a bound does not exist, the source state nodes are unbounded anyway, so model checking would be impossible anyway.

8 Dynamic Concurrency

According to the UML definition [23], it is possible to have “dynamic concurrency” in an action state node (or subactivity state node). This means that the corresponding activity (or activities) is instantiated multiple times in parallel, denoted by marking the action state node (or subactivity state node) with ‘*’. The word ‘dynamic’ indicates that the number of instantiations is determined at run-time. The word ‘concurrency’ indicates that the instantiations execute in parallel. In Figure 17 an example of dynamic concurrency within an action state node is given (adapted from Fowler [13]). First, an order is received. Next,

the order is filled with each line item on the order. Finally, the order is delivered. The number of instantiations of *Fill line item* is dynamic (i.e., determined at run-time), since this number depends on the order that is being processed.

We now sketch how dynamic concurrency can be formalised for action state nodes and subactivity state nodes.

Action state nodes. We assume a set $DS \subseteq AS$ of atomic action state nodes whose activities are dynamically instantiated. Every action state node a in DS is annotated with a dynamic concurrency expression $*[expr]$. We assume $*[expr]$ evaluates to a list of elements. So $*[expr]$ is of type list. If a is entered, for every element e in list $[expr]$ an activity $control(a)_e$ is started. If all started activities are completed, a terminates. So action state node a now represents the execution of a dynamically determined number of activities, rather than one.

We do not have to change our definition of configuration and steps, since for both an action state node with dynamic concurrency and an ordinary action state node, the state node is part of the configuration and the state node terminates. Merely the function *control* needs to be redefined: it is not static anymore, but dynamic: the activities assigned to an action state node that has dynamic concurrency are defined when the dynamic concurrency expression is evaluated, i.e. when the state node is entered.

Subactivity state nodes. Remember that we eliminated subactivity state nodes from an activity diagram by substituting the activity diagram specification of the subactivity state node. If the subactivity state node contains a dynamic concurrency expression $*[expr]$, we treat the activity diagram specification as a parametrised specification. Assuming $*[expr]$ evaluates to a list of elements, the activity diagram specification is copied once for every possible element of the list. For each copy, each state node of the copy is subscripted with the name of the element; so, different copies have unique state nodes. All these copies are started at the same time. Hence, in the initial state of the activity diagram specification, parallel branches are started. Which branches are taken, and which not, depends on the evaluation of $*[expr]$. The decision is a simple test whether an element belongs to the list or not: if so, the branch is taken, otherwise it is skipped.

We illustrate this by a simple example. Let subactivity state node **B** in Figure 18 have a dynamic concurrency expression that evaluates into a list of natural numbers. Let this dynamic concurrency expression depend on local variable x , where x has type natural number, set by activity **A**. If **B** is specified as in Figure 19, then the activity hypergraph becomes as depicted in Figure 20. Note that there are infinitely many copies made of the activity diagram specifying **B**, since there are infinitely many natural numbers.

9 Related Work

Of course, our semantics is set up in such a way that it satisfies our semantic requirements. In this section we study whether for UML activity diagrams (a.o. UML statecharts and Petri nets) and other semantics for workflow modelling (Petri nets, STATEMATE statecharts) satisfy our semantic requirements. The results are summarised in Table 3.



Figure 17: Example: multitask

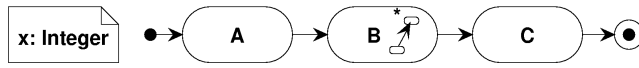


Figure 18: Dynamic concurrency example

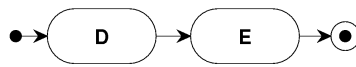


Figure 19: Specification of *B*

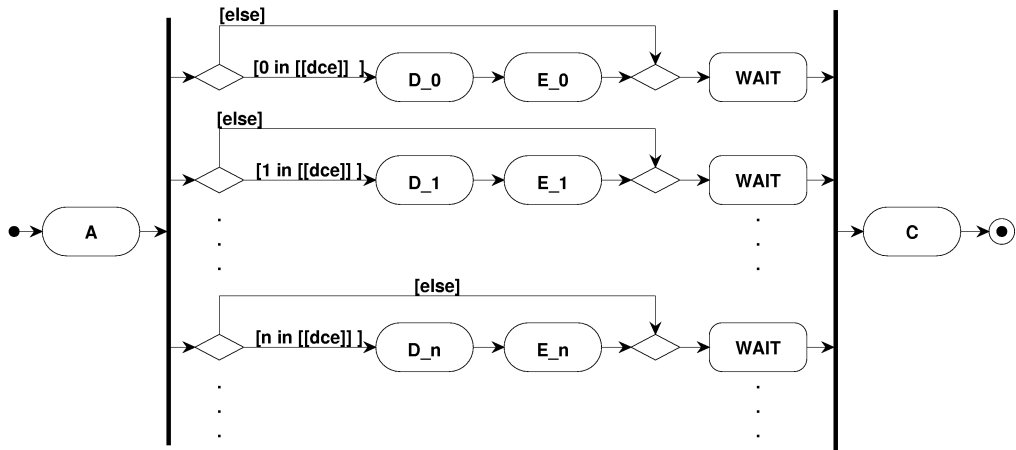


Figure 20: Dynamic concurrency

requirement	we	Petri nets	STATEMATE statecharts	UML statecharts
1	+	-	+	+
2	+	+	+	-
3	+	±	+	-
4	+	-	-	-
5	+	-	-	-
6	+	+	-	+

Table 3: Requirements satisfied by other semantics

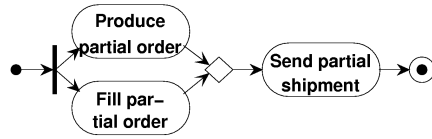


Figure 21: Activity diagram that cannot be translated into a statechart (repeats Fig. 1)

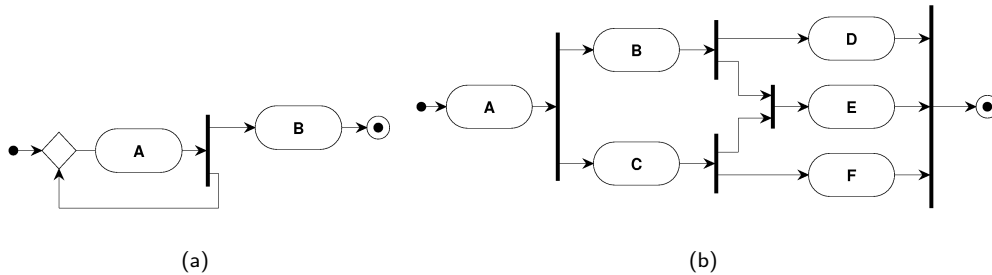


Figure 22: Other activity diagrams that cannot be translated into a statechart

9.1 Semantics for UML activity diagrams

Statechart semantics of activity diagrams. The UML definition [23] gives a semantics to a UML activity diagram by considering it as a special kind of UML statechart. Statecharts were introduced by Harel [15] to model behaviour of activities in the structured analysis notation STATEMATE [18] (see below). The statechart notation was widely adopted by OO people, and is now part of the UML [23].

Statecharts in general model parallelism using a hierarchy of state nodes. This hierarchy must be a tree, so that a state node cannot have more than one parent. There are two kinds of hierarchical state nodes: AND state nodes and OR state nodes. A statechart configuration must satisfy amongst others the following constraints:

- if an AND state node is in the configuration, all its immediate substate nodes (children) must be in the configuration too,
- if an OR state node is in the configuration, precisely one of its immediate substate nodes (children) must be in the configuration too.

A configuration is because of these constraints a set of state nodes (whereas in UML activity diagrams configurations are bags) and steps are also sets (in UML activity diagrams bags). These hierarchical constraints limit the concurrency of a statechart in such a way that it is less expressive than an activity diagram. Our running example can only be translated into a statechart, by introducing some extra states (see below). Other activity diagrams are even impossible to express in statecharts. Consider for example Fig. 21 (see Section 2 for an explanation of this example). In Figure 22(a), another activity diagram is shown that cannot be translated into a statechart. Activity state node B cannot be put in the statechart hierarchy, since it executes both in parallel with A (which should be modelled by an AND state) and after A (which should be modelled by an OR state). In fact, multiple copies of

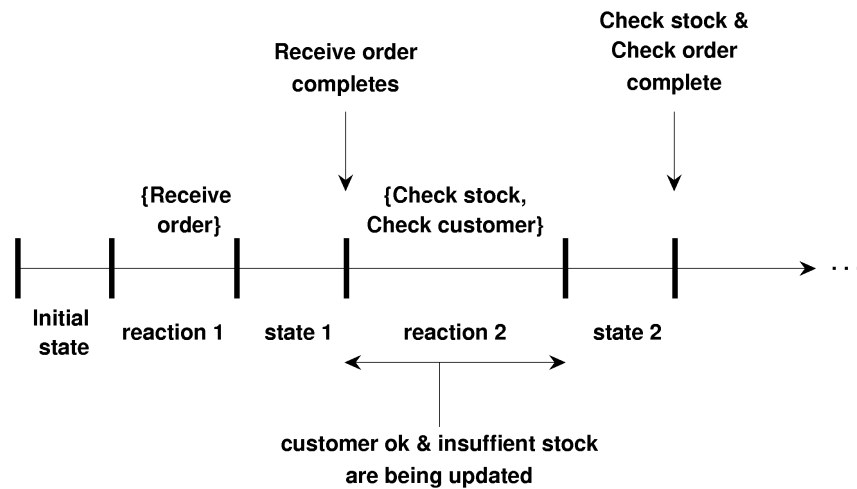


Figure 23: Run of activity diagram of Figure 2 according to the OMG semantics

B may be active at the same time. Since in a statechart at most one copy of B can be active, this activity diagram cannot be modelled as a statechart. Yet, the activity diagram in Figure 22(a) is allowed by the UML syntax and it represents a workflow that can occur in practice. Another example is Figure 22(b). In this example, activity state node E cannot be modelled in the statechart hierarchy for the following reason. Node E belongs to both the OR state node containing B and the OR state node containing C. Both these OR state nodes are in parallel, i.e., belong to the some AND state node, and are not hierarchically related, since B and C are in parallel. Therefore, node E must have two parents, which is forbidden in statecharts.

In the OMG activity diagram semantics, atomic activities are mapped to actions in a statechart (hence the name action state node for an atomic activity state node). Therefore, the following semantic requirements are not satisfied. Since actions take time, the perfect technology assumption is violated (2) and events are not responded to when they occur (3) (since an action may be busy executing). A statechart action is performed in a transition of the statechart. This implies for workflow models that the WFS updates the case attributes, instead of actors (4). No integrity constraints for local variables are specified (5). Finally, as noted above, activity diagrams are more expressive and more concise than statecharts.

Moreover, this semantics of activities leads to a counter-intuitive execution structure since a dependency between activities is introduced. Figure 23 shows the start of all runs of the activity diagram of Figure 2, obtained by interpretation of the informal OMG semantics. In this semantics, both **Check stock** and **Check customer** must stop at the same time, whereas in Figure 2 we do not want to specify this. Thus, they are made dependent on each other. Another dependency is introduced by the fact that every activity now has to last at least as long as the activities that are executed in parallel with it.

Finally, the OMG semantics is defined at the software implementation level, whereas we want to stay at the requirements level for WFSs, not for software. In the run above (Fig. 23), both states and reactions take time, and it also takes time to process events. Since reactions are non-instantaneous, during a reaction the system is locked. Consequently, this semantics

allows the response to some (temporal) events to be too late. This mixes requirements with implementation restrictions.

We conclude that the OMG semantics may be suitable to represent the execution of OO programs, but not the execution of WF systems.

Petri net semantics of activity diagrams. Gehrke *et al.* [14] give a Petri net semantics to UML activity diagrams. They map state nodes to places, and edges to transitions. The differences with our semantics are the following.

First, they do not formally define data or real time, although they give some suggestions on how guards could be modelled. This makes a detailed comparison with their work difficult. In particular, it is not defined where and when case attributes are updated: in steps or in states. Also, they do not define a formal execution semantics; probably, they use the standard Petri net interleaving or step semantics [19, 22]. Again, this makes a comparison with our semantics difficult. We assume the step semantics here, since it resembles most our own semantics. Figure 25 shows part of a run satisfying the Petri net step semantics. This part of the run would also satisfy our semantics, if we add information on how the environment behaves (cf. Fig. 12). Note that as in our semantics steps are instantaneous (Petri nets satisfy perfect technology by definition since transitions fire instantaneously).

Second, Petri net semantics do not model input events. Firing of a transition is determined by the current Petri net configuration, and this does not contain input events. (Semantic requirement 2.) One might try to get around this by defining the firing of a transition as the occurrence of an input event. For example, one could try to define the Petri net semantics of Fig. 24(a) to be Fig. 24(b), where e is an input event. But the intended meaning of Fig. 24(a) is that in the indicated configuration [WAIT, WAIT], input event e triggers two transitions simultaneously to configuration [A, B]. And as we just observed, in Fig. 24(b), transitions are determined by the presence of tokens in places, and not by the presence of input events. So in Fig. 24(b), the transition from place WAIT to place B can be taken independently from the transition from place WAIT to place A. That would correspond to two independent arrivals of input event e and this is not what we intended with Fig. 24(a) at all. If we would restrict ourselves to an interleaving semantics, this observation no longer holds. But an interleaving semantics is not appropriate for workflow models, since it assumes that only one edge can be taken at a time, so a step only contains a single edge. As explained in Section 3, the step should be maximal: the WFS should take as many transitions as possible in one step. We conclude that we cannot incorporate input events in Petri nets by simply labelling transitions with input events.

Another possible way to model input events in Petri nets is to model the occurrences of input events as tokens. For each input event a place is made. If this place is filled with a token, the event occurs, otherwise it does not occur. One drawback of the event-as-token approach is that in our semantics more than one transition can be triggered by the same event e . To simulate such behaviour in a Petri net in this way, more than one e token is needed. However, the exact number of tokens for e is not known beforehand. We therefore would have to introduce a lot of spare tokens. This invalidates existing verification techniques for Petri nets, e.g. the ones introduced by Van der Aalst [2]. As above, this observation does not hold for an interleaving semantics, because then one token per event place would suffice. As explained above we do not consider an interleaving semantics appropriate. Another drawback of the event-as-token approach is that the resulting Petri net looks like ravioli, since the place where the input token e resides must be connected to all transitions that are

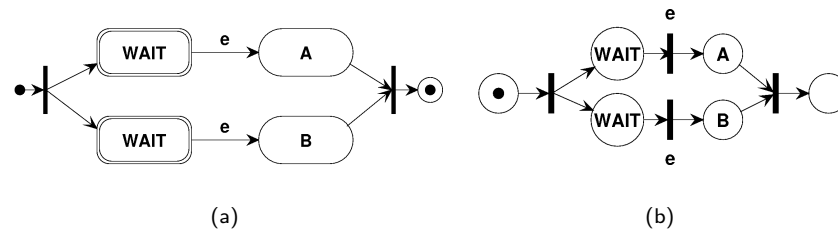


Figure 24: Example to illustrate the difference between Petri net semantics and our semantics

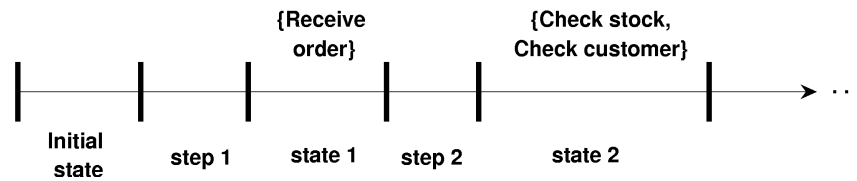


Figure 25: Petri net execution: activities are places

triggered by e . We conclude that input events also cannot be included in Petri net semantics by using event places and event tokens.

Third, in our semantics case attributes are updated in a state (so by the actors, semantic requirement 4). This seems impossible to model in Petri nets, since in Petri nets all actions, including attribute updates, must be done by transitions.

Fourth, some constructs we (with the UML) consider syntactic sugarring, they consider as real syntax. (They model pseudo state nodes as real state nodes.)

Concludingly, the semantics of Gehrke *et al.* [14] is not fully defined. To extend it to cover all the features that we deal with, the following features should be added to the semantics:

- inputs, to model the environment,
- clocks, to model real time,
- local variables, to model case attributes,
- activities, in which case attributes are updated by actors.

To the best of our knowledge, there does not exist a Petri net semantics having these features. One could therefore also see our semantics as the first semantics for reactive Petri nets.

Note: In the next subsection we explain why we require local variables, and not coloured tokens, which is by now the standard way to incorporate data into Petri nets. **End of note**

Abstract State Machine semantics of activity diagrams. Börger *et al.* [6] given an abstract state machine (ASM) semantics to activity diagrams. In an abstract state machine, a state is an algebra and if-then rules define how to go from one state to another. The differences with our semantics are as follows. First, the ASM semantics tries to stick as close

as possible to the OMG semantics [23]. The ASM semantics is therefore an implementation level semantics. The run shown for the OMG semantics (see Figure 23) is also a run in ASM semantics. Second, the ASM semantics is not real time. Third, the ASM semantics presupposes nested forks and joins, i.e., every fork should be followed by a subsequent join that joins all fork branches. Although the UML suggests this nesting, it is not obliged by UML and we therefore did not demand it. Besides, we think that not every valid activity diagram has to satisfy this nesting rule (look at our examples in Fig. 2, 22(a) and 22(b)). Fourth, in the ASM semantics the state of the activity diagram is a set of edges, rather than a set of nodes as it is in our semantics. Also, in the ASM semantics state changes are represented by nodes of the activity diagram. This swapping of state and state change w.r.t. to the original UML definition leads to a non-intuitive interpretation of an activity diagram, since a state node is mapped to a transition and an edge to a state.

Furthermore, in the ASM semantics an activity is performed in a system reaction. Therefore, parallel activities are made dependent on each other, just as in the OMG semantics. We have seen earlier that this is undesirable.

Besides, the ASM semantics considers an activity to be part of the WFS, rather than part of the environment (just as the OMG semantics of UML does). This would mean for workflows that updates to case attributes are made by the WFS, not by the environment. Finally, a minor difference is that pseudo state nodes are treated as just like ordinary state nodes, although the authors claim to follow the OMG semantics, which does not treat pseudo state nodes as ordinary state nodes.

We conclude that the ASM semantics is inappropriate for workflow modelling since it sticks to the OMG semantics (although not completely), it therefore is an implementation-level semantics, and it presupposes nested forks and joins. Furthermore, the definition of state is counter-intuitive to what is suggested by the activity diagram itself.

9.2 Other formal workflow languages

Petri nets. A variety of Petri net variants has been conceived for workflow models, amongst others by Van der Aalst [2] and Ellis and Nutt [10]. Since ordinary Petri nets are not powerful enough to model workflows, these authors use high-level Petri nets, which are Petri nets extended with time, data and hierarchy. In the following discussion, we focus on the high-level Petri net model of Van der Aalst [2].

Van der Aalst [2] uses interval timed coloured Petri nets [1] to model workflows. Interval timed coloured Petri nets are ordinary Petri nets extended with coloured tokens (to model the data) and timing intervals associated to transitions (to model duration of transitions). Figure 26 shows our example in high-level Petri net notation. Circles represent places. Rectangles represent either activities, AND-splits, AND-joins, or OR-splits. Since ordinary Petri net transitions are forks, filling all output places rather than one, a decision transition is explicitly represented by a special symbol. In our example, we simply write **OR-split**. (Note however that high-level Petri nets *can* model choice since not every output place of a transition needs to be filled.) A global state (called marking in Petri net theory) is represented by distributing tokens over places. A token denotes that a state is active.

The semantics for executing a single transition of an interval timed coloured Petri net is as follows. Assume a global clock that measures the current time. Every token is time-stamped with the time the token becomes available for firing. A single transition fires as soon as the current time is equal to the maximum time stamp on its input tokens, since then all

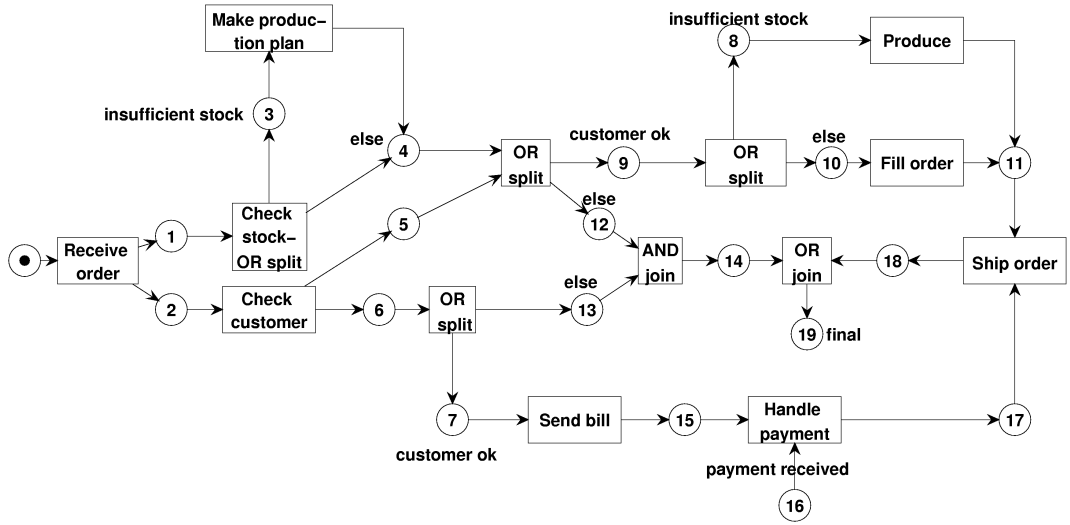


Figure 26: Petri net model of our running example

input tokens are available according to their time-stamps. The transition then immediately fills all its output places with a time-stamped output token. All output tokens will have the same time-stamp that must be greater than or equal to the time of firing of the transition. The difference between the time-stamp of the output tokens and the time of firing, called firing delay, must be in the interval attached to the transition. The firing delay represents the duration of the transition.

Van der Aalst [2] does not provide an explicit execution semantics; we assume here the Petri net step semantics [25], since it most resembles our semantics. We repeat the remark made above for the Petri net semantics of UML activity diagrams: Petri nets semantics do not model the environment: the step taken only depends upon the current configuration, not on the set of input events. We make the assumption that or-splits and or-joins and and-joins do not take time, whereas transitions corresponding to the execution of activities do take time (we do not specify precise intervals).

Figure 27 shows the execution of the example according to the Petri net semantics. A state is a configuration (called a marking in Petri net theory). In this case, a configuration is a multiset of time-stamped tokens on places. Steps are sets of transitions in the Petri net.

We now discuss the differences with our semantics by looking at the list of semantic requirements. (1) In the high-level Petri net semantics, the environment is not modelled: Firing of a transition depends only upon the current marking (including the time-stamps of the tokens in the marking). (2) The high-level Petri net semantics satisfies perfect technology, since transitions fire instantaneously. Note: a transition in this model actually represents the *starting* of an activity, rather than the *execution* of an activity (which is suggested by the model). (3) One could argue that the high-level Petri net semantics has some kind of clock-asynchronous semantics, since a transition fires as soon as its enabled, which is also true in the clock-asynchronous semantics.

Intermezzo: In high-level Petri nets, there exists a concept that is similar to our notion of a pseudo state node. If we assume that or-splits, or-joins, and and-joins do not take time,

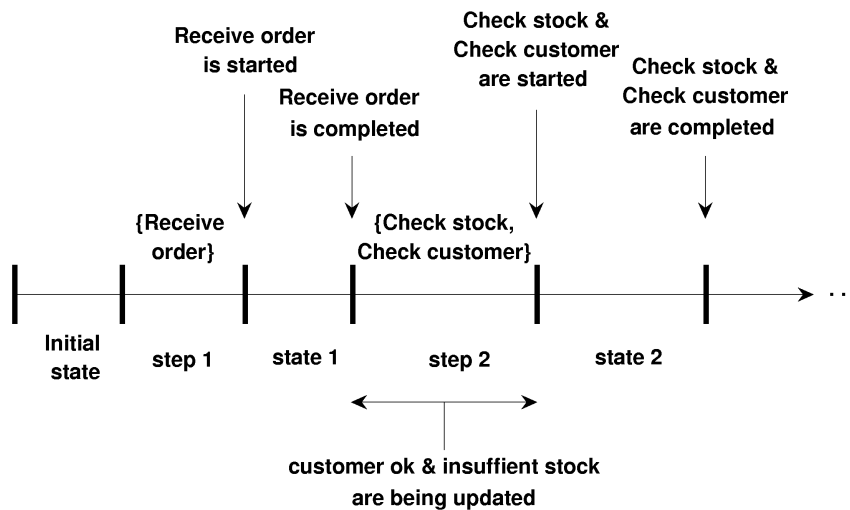


Figure 27: Petri net execution: activities correspond to transitions

any token fired by any of these transitions is immediately available for the next transition. This means that after an or split, or join, or and join fired, immediately some other transition may fire. (This is called a vanishing marking in stochastic Petri net theory [4]).

We illustrate the similarities between our clock-asynchronous semantics and the high-level Petri net semantics by means of our running example in high-level Petri net notation (Fig. 26). For example, assume the current configuration is $[4,5,6]$ (representing that Check stock and Check customer are busy executing, and that there is sufficient stock, and assume that both activities Check stock and Check customer complete at the same time. If the decision customer ok is made, a possible next configuration will be $[7,9]$. This configuration is however unstable, since both transitions or split (having place 7 as input place) and Send bill are enabled and thus fire. A possible next configuration is $[10,15]$ (remember that there is sufficient stock). This configuration is also unstable, because the transition Fill order is enabled. The next configuration, which is stable, is $[15,11]$, representing that Fill order and Send bill are busy executing. This execution is similar to the one made in our semantics: if in the activity diagram model (Fig. 2), the current configuration is Check stock, Check customer and both activities complete at the same time and there is sufficient stock and the customer is ok, the next configuration is Fill order, Send bill (shown in Table 2). **End of intermezzo**

On the other hand, since the environment is not modelled, the question whether it reacts immediately to events from outside cannot be answered in the model. (4) Although case attributes are updated in activities, these activities are under control of the Petri net itself (so the WFS), instead of the environment (the actors). As we explained in Sect. 3, we feel this is inappropriate. In particular are post conditions not specified declaratively, but imperatively. (5) In high-level Petri nets, there exist no data integrity constraints. Case attributes are represented by coloured tokens. By centralising tokens in separate places, one can enforce data integrity, however at the cost of reduced concurrency (our data integrity constraints allow for more concurrency). (6) In high-level Petri nets, it can be modelled that

an activity takes time.

Van der Aalst [2] does provide a notation for attaching a trigger event (including a temporal one) to an activity. However, he does not provide a semantics for this notation, but abstracts away from it for two reasons. His first argument is that the environment cannot be modelled completely. In our opinion, this argument seems to point out a lack of modelling capabilities in Petri nets that is similar to the ones identified by us above. His second argument is that if an abstracted workflow is correct, the concrete one will also be. In our opinion, this second argument can only be proven by giving a semantics to the notation and then comparing the abstract and concrete workflow models (i.e. proving some kind of equivalence). But this is not done by Van der Aalst.

Minor point: A choice (OR-split) is modelled by a transition. When the transition fires, the choice is made. This has awkward consequences, if a transition is an activity combined with an OR-split. Translating this to workflow models, it means that the choice is made not when an activity has completed, but when it is started. For example, if in Fig. 26, transition **Check stock** fires, immediately a token is put in one of the two places 3 or 4 (but this token is not available for the next transition to fire, because the time stamp is greater than the current time, since the activity takes time to execute). This awkward situation can be circumvented by not combining an or-split with a transition that represents an activity, but putting the or-split immediately after the transition that represents an activity. **End of minor point**

Summarising: Petri nets fail to model the environment. A rather surprising conclusion is therefore that although Petri nets are widely used to specify workflow models, according to our semantic requirements, they are inappropriate for workflow modelling for two reasons. First, our requirements are motivated by our view of a workflow system as a reactive system. Key property of a reactive system is its interaction with the environment: the WFS is an open system. A Petri net, on the other hand, only models a non-reactive, closed system. A Petri net has no interaction with its environment, since this environment simply does not exist. We suspect that most Petri net based workflow packages have only adopted the notation, but not the execution semantics of Petri nets.

An inspection of one of the leading workflow management tools [20] reveals that, although the notation of the workflow model is based on Petri nets, the only similarity between the implemented semantics and the Petri net semantics is that only one transition can fire (i.e. only one activity can complete at a time), which seems to us an unnecessary implementation restriction. We assume that as many transitions as possible are taken (i.e. a step is maximal). What happens when a transition fires (navigation) is formalised in [20], but the behaviour of the environment is not (because the purpose of the authors is not verification of workflow models).

A second observation is that case attributes are difficult to model in Petri nets. To have data integrity in a Petri net model, for each case attribute, the Petri net should contain a separate place that holds its current value. Each activity that reads or updates the attribute should be connected to this place by a transition. This leads to a ravioli diagram that is difficult to understand and maintain. Our proposed activity diagram semantics does not have these drawbacks.

Statestate statecharts. Another formal modelling technique, not as popular as Petri nets for workflow modelling, is statecharts. Statecharts were introduced by Harel [15] to model behaviour of activities in the structured analysis notation STATEMATE [18]. This

notation was soon adopted by OO people, and it is now part of the UML [23]. Its popularity stems from the efficient representation of parallelism, thereby avoiding the state-explosion problem.

Wodtke and Weikum [27] use STATEMATE statecharts and activity charts [18] to model (distributed) workflows. An activity chart, not to be confused with a UML activity diagram, describes the dataflows between activities. An activity chart is controlled by a statechart, that can start and stop activities. The activities themselves are not declaratively specified. Statecharts specify the control flow between activities. Figure 28 shows the statechart model of the example. A final state node is denoted by a circle, labelled by a T (from terminal). Note that in some state nodes upon entry an activity, with the same name as the state node, is started. These state nodes are similar to our action state nodes. If the activity completes, the corresponding state node is left. Note further the introduction of compound state nodes (state nodes containing other state nodes) to model parallelism. Finally, note that we had to express the synchronisation of WAIT-1 and WAIT-2. We could not use a hyperedge with source {WAIT-1,WAIT-2}, instead of the single edge labelled with the in predicate, since if that hyperedge would be taken, state WAIT-2 would be left (and thus state F1), whereas F2 would not be left. Since not all children of the AND state F12 would be active, the resulting configuration would be invalid. This example shows the limitation of statecharts in expressing synchronisation between parallel branches.

For this example, a run of the statechart coincides with the one of the activity hypergraph (see e.g. Figure 29). We assume that each activity is self-terminating, and thus not stopped by its controlling statechart. States and (super)steps in the run of the statechart are similar to the states and (super)steps in the run of an activity hypergraph.

There are two important differences with the UML Statecharts. First, the priority definition of STATEMATE is the reverse of the UML. In STATEMATE a high level transition has priority over a lower level one, whereas in the UML, it is the other way around. For example, if subactivity A_1 of non-atomic activity A completes, the transition emanating from A will have precedence over the one emanating from A_1 , and the execution of A is interrupted. In the UML, it is the other way around. Because of this, we had to label the transitions outgoing Processing order, whereas this would not have been necessary in UML.

Second, a final state node in a STATEMATE statechart denotes termination of the whole statechart, whereas in an UML statecharts, it denotes termination of the super state. Hence, we did not put any terminal state nodes in place of Production done and Finance done.

Although our semantics is based on the STATEMATE statechart semantics, there are some differences w.r.t. the semantic requirements. (4) We specify activities declaratively with pre and post-conditions (whereas in STATEMATE they are specified imperatively with a procedure or a statechart). Moreover, activities in STATEMATE are under control of the system, not of the environment. (5) In addition, we have defined data integrity constraints that do not exist for STATEMATE. (6) Next, activities can only be part of the state in STATEMATE, if they are specified imperatively by a statechart, which is inappropriate.

Some other differences are the following. First, statecharts have constrained parallelism, cannot model unboundedness and have limitations in expressing synchronisation. So, activity diagrams are more expressive than statecharts. Second, the STATEMATE statechart semantics uses discrete time, whereas we use continuous time.

We conclude that since in STATEMATE data is updated by the system itself, not by the environment, STATEMATE statecharts are not appropriate for workflow modelling. In addition, STATEMATE does not allow the specification of data integrity constraints, needed

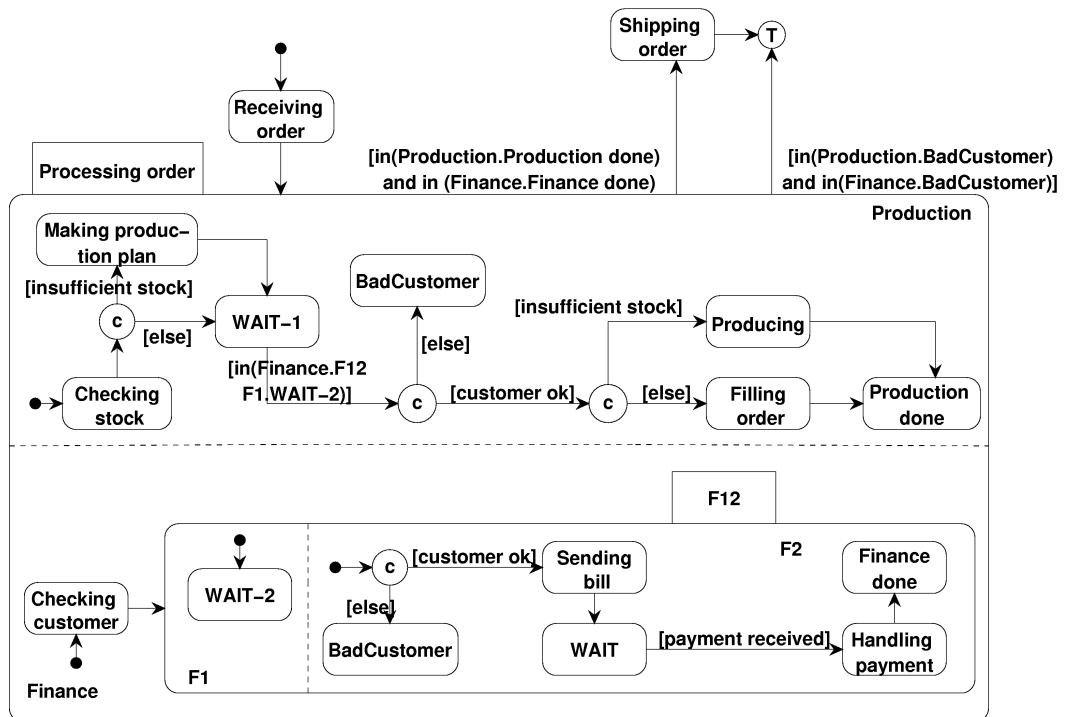


Figure 28: STATEMATE statechart model of our running example

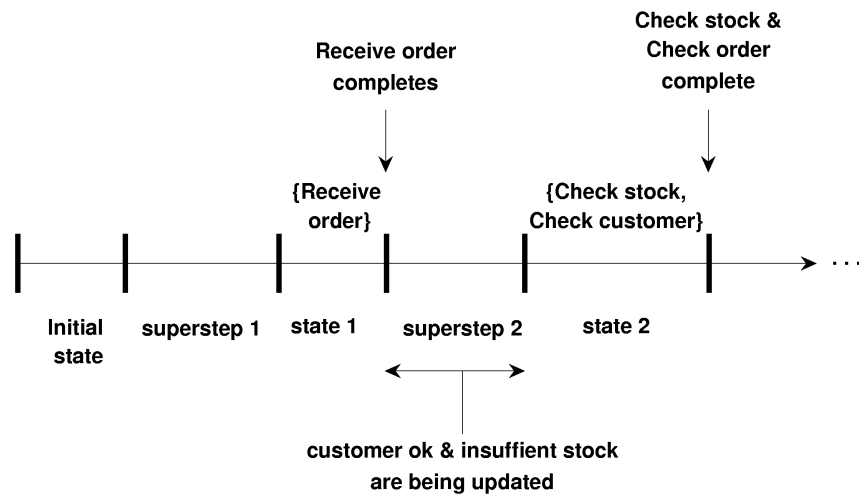


Figure 29: STATEMATE statechart execution of our running example

for workflow modelling. Finally, because super state nodes must be shown, and final state nodes cannot be used to model termination of an OR state node, statecharts are not as concise in representing a workflow as an activity diagram.

10 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 statechart and Petri net semantics, extended with transactional properties. We defined both an execution and a transition system semantics. Our semantics is motivated by analysis of the workflow literature and by case studies. Our semantics is different from other proposed semantics, both for activity diagrams and for workflow models. Because of the arguments in Section 3, we believe our semantics is appropriate for workflow modelling. To substantiate this point further, we are currently implementing the semantics as an execution facility to the CASE tool TCM [8]. We will validate our semantics by applying it to several case studies that we did. In addition, we are implementing an interface between TCM and Kronos [29], to facilitate model checking of properties of activity diagrams.

Future work includes the following. The work presented in this report can be easily extended to model cross-organisational workflows. We then have to model in addition *parties*. Each party executes its own part of a workflow. Next, we have to model communication between parties. Finally, we have to redefine our nonconflict constraint. Each party will have its own local database for which the nonconflict constraint must hold, but *between* parties, so globally, this constraint does not have to hold, since there is no global database.

Another useful extension is to define a pattern language for workflow patterns. This language must contain operators to combine workflow patterns. Next, the patterns need to be checked for consistency.

An interesting topic is to apply our model checking semantics to the domain of transactional workflows. Model checking seems to have an advantage over Petri net based verification, done by Derks *et al.* [9], since we can model the transactional properties as formula's, rather than as activity diagram or Petri net, which leads to a simpler and elegant solution to the problems posed by Derks *et al.*

References

- [1] W.M.P. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer-Verlag, 1993.
- [2] 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.
- [3] 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>.
- [4] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems*. Computer Systems Series. MIT Press, 1986.
- [5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [6] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In Teodor Rus, editor, *Proc. Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.
- [7] P.R. D’Argenio. Personal communication, 2000.
- [8] 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>.
- [9] W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflows. Technical Report TR-CTIT-00-24, University of Twente, 2000.
- [10] C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993.
- [11] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamins/Cummings, Redwood City, 1989.
- [12] 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.
- [13] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Modeling Object Language*. Object Technology Series. Addison-Wesley, 1997.
- [14] 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, Institut für Informatik, Universität Hildesheim, 1998.
- [15] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [16] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [17] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO, ASI Series*, pages 447–498. Springer-Verlag, New York, 1985.
- [18] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts : the STATEMATE approach*. McGraw-Hill, 1998.

- [19] K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, 1992.
- [20] F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques*. Prentice Hall, 2000.
- [21] S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, New York, New York, 1984.
- [22] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
- [23] OMG. *Unified Modeling Language version 1.3*. OMG, July 1999.
- [24] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1981.
- [25] W. Reisig. *Petri Nets (an Introduction)*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.
- [26] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
- [27] 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)*, volume 1186 of *Lecture Notes in Computer Science*, pages 230–246. Springer, 1997.
- [28] Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. <http://www.wfmc.org>.
- [29] S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, 1997.