Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets

Rik Eshuis* Roel Wieringa

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

Abstract. Petri net variants are widely used as a workflow modelling technique. Recently, UML activity diagrams have been used for the same purpose, even though the syntax and semantics of activity diagrams has not been yet fully worked out. Nevertheless, activity diagrams seem very similar to Petri nets and on the surface, one may think that they are variants of each other. To substantiate or deny this claim, we need to formalise the intended semantics of activity diagrams and then compare this with various Petri net semantics. In previous papers we have defined two formal semantics for UML activity diagrams that are intended for workflow modelling. In this paper, we discuss the design choices that underlie these two semantics and investigate whether these design choices can be met in low-level and high-level Petri net semantics. We argue that the main difference between the Petri net semantics and our semantics of UML activity diagrams is that the Petri net semantics models resource usage of closed, active systems that are non-reactive, whereas our semantics of UML activity diagrams models open, reactive systems. Since workflow systems are open, reactive systems, we conclude that Petri nets cannot model workflows accurately, unless they are extended with a syntax and semantics for reactivity.

1 Introduction

Petri nets are a popular technique for modelling the control flow dimension of workflows. When modelling workflows, people tend to draw nodes that represent tasks or activities, and arrows between the nodes that represent sequencing of activities. The resulting diagrams look like Petri nets, and so Petri nets seem a natural technique for modelling workflows [2,22]. The following arguments are often used to support this: Petri nets are graphical, they have a formal semantics, they can express most of the desirable routing constructs, there is an abundance of analysis techniques for proving properties about them, and finally they are vendor-independent. Most of these arguments do not refer to the domain of workflow modelling (only the routing argument does) and point out advantages of Petri nets in general. Moreover, since Petri nets already existed before

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

workflow management systems were invented, their semantics is not specifically intended for workflow modelling. So none of these arguments state why and how Petri nets are useful for workflow modelling. This is unsatisfactory for analysis purposes, since analysing a Petri net workflow model presupposes that the Petri net models the real workflow accurately.

Recently, UML activity diagrams [56] have also been used for workflow modelling. They too are graphical, use bubbles and arrows, are vendor-independent, and can express most desirable routing constructs. Unfortunately, the OMG semantics [56] is not formal (nor precise), and it is not intended for workflow modelling [28]. We therefore defined two semantics for UML activity diagrams that are intended for workflow modelling [27,28]. The goal is to use these semantics for analysing workflow models in activity diagram notation by means of model checking [16,25]. The first semantics is a high-level semantics, based upon the STATEMATE semantics of statecharts [37], that is easy to analyse (both for a computer and for a person) but somewhat abstract. By contrast, the second semantics is low-level and resembles both the behaviour of an abstract workflow system and the informal OMG semantics of UML state machines, but it is more difficult to analyse than the first semantics. We have implemented verification support using model checking for the first semantics in our diagram editing tool TCM [17,25].

In this paper we discuss the design choices that underlie both our formal execution semantics. Since our purpose is to make analysis of activity diagram workflow models possible, the semantics must be an accurate representation of workflow behaviour. Our design choices are therefore motivated in terms of the domain of workflow modelling. Using these choices as a yard stick, we investigate how well Petri nets can model some important aspects of workflow modelling. We hope this provides relevant arguments for and against the claim that Petri nets are useful for workflow modelling.

This approach may seem subjective, since other persons might make other design choices, and consequently they might draw other conclusions about the suitability of Petri nets for workflow modelling. However, we think that the choices we have made in our semantics are reasonable, because they are motivated by the domain of workflow modelling. Even if one does not agree with the choices we made, our discussion gives – we hope – more insight in possible answers to the question what actually is a Petri net [20].

Our most important design choice is that the semantics for activity diagrams must be reactive. The token-game semantics, which is characteristic for Petri nets, does not represent reactivity, which is characteristic of workflow systems. A Petri net transition can fire if all its input places are in the current marking [48,51]. But in a reactive system a transition can be taken (fired) if all its source nodes (input places) are in the current configuration (marking) *and* its trigger event occurs [37,56]. This trigger event is an event in the environment of the system, that the system will react to by taking the transition. Although Petri nets in our view are not reactive, we will study different ways of *simulating* reactive behaviour in different Petri net variants.

In the sequel, we presuppose some basic knowledge of Petri nets and high-level Petri nets (see e.g. [40,48,51,53,54]). We have looked at Petri net variants that are traditionally used to model and analyse workflows, namely Workflow Nets [2,3], Information Control Nets [22], INCOME/WF [49], FunSoft nets [18], MILANO WFMS [9]. Next, we have looked at Petri net variants that are not specifically tailored towards workflow modelling but nevertheless can be useful: Open Nets [10], Petri nets with synchronous communication [15], Signal-Event Nets [36,29], Contextual Nets [47,30], Zero-Safe nets [12], and several variants of Object-Oriented Petri Nets [8,46]. More information about some of these references can be found in recent overviews and collections about the use of Petri nets for workflow modelling [1,55]. A comparison of our semantics with other formal modelling techniques (in particular STATEMATE [37,59]) can be found elsewhere [26].

Structure. We start by explaining some characteristics of workflows and workflow systems in more detail. In Section 3 we discuss our two activity diagram semantics and the design choices that underlie these semantics. We also discuss the properties of every basic statechart step semantics, used in both STATEMATE [37] and UML [56]. We have adopted these properties as well in both our formal semantics. In Section 4 we study whether and how our semantics can be simulated in Petri nets. In particularly, we discuss whether and how the statechart step semantics can be modelled in Petri nets. We end with conclusions.

2 Workflow

This section is based on literature (amongst others [2,45,60]) and several case studies that we did. 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 (also known as 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. An example of a case is the process that handles the insurance claim of John Smith. An example of a work item is the claim form of John Smith. 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 control-flow dimension and the resource dimension [2,45]. The control-flow dimension concerns the ordering of activities (or tasks) in time (what has to be done). The resource dimension concerns the organisational structure (who has to do it). Since both Petri nets and UML activity diagrams only model the control-flow dimension, we here focus on modelling the control-flow dimension of workflows. When we use the term workflow model, we refer to a model that describes the control-flow dimension.

Activities are done by *actors*. An *activity* is an amount of work that is uninterruptible and 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

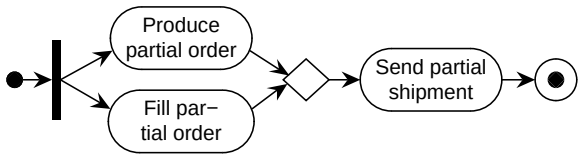

Fig. 1. Multiple simultaneous instantiations of Send partial shipment

relevant data. The case may be distributed over several actors. Each distributed part of the case has a *local state*. There are two 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. A special event is that the actor who has to do the next activity 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 notation is explained in the next section). 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.

Actors are people or machines. Actors are grouped according to roles. A *role* is a set of characteristics of actors. A role can refer to skills, responsibility, or authority for people, and it can refer to computing capabilities for machines [45,61]. Roles link actors and activities. The modelling of actors and roles, and the connection with workflow models falls outside the scope of this paper.

The *effect* of an activity can be constrained declaratively with a pre- and post-condition. The effect cannot be specified fully since execution of the activity falls outside the scope of the WFMS. The pre-condition also functions as guard: as long as it is false, the activity cannot be performed.

The WFMC [61] specifies four possible ordering relationships between activities: *sequence*, *choice*, *parallelism* and *iteration*. Van der Aalst et al. identified more ordering relationships [7]. 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*.

Architecture (Fig. 2). The following architecture is based upon amongst others [14,34,45,60]. A workflow system (WFS), which is a WFMS instantiated with one or more workflow models, connects a database system and several applications that are used by actors to do work for the cases. In this paper we

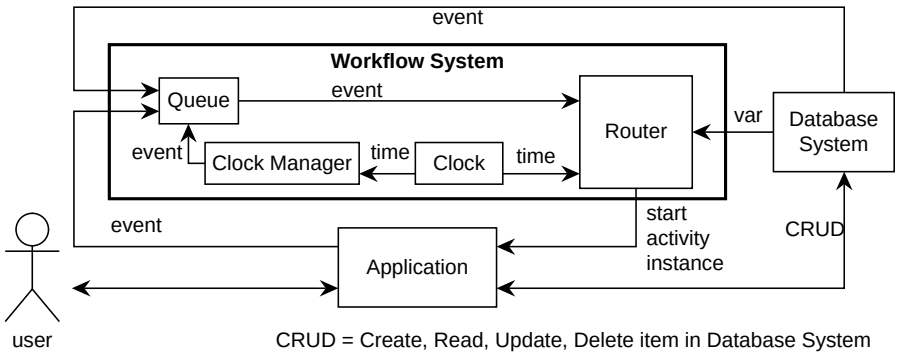

Fig. 2. Abstract workflow system architecture

assume that the WFS controls a single case (a generalisation to a WFS that controls multiple cases is straightforward). The main components of the WFS are the queue and the router. The environment interacts with the WFS by putting events in the queue. On basis of these events and the current state of the case, the router component of the WFS routes the case as prescribed by the workflow model of the case. As a result, some new activity instances can be started. 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. By taking these transitions the WFS routes the case. All attributes of a case are stored in the database. The state of the case is maintained by the WFS itself. Scheduled timeouts are maintained and raised by the clock manager on basis of the internal clock.

3 UML Activity Diagrams

Syntax. We explain the syntax by means of a small example. In Fig. 3 the workflow of "Processing Complaints" is shown (converted from a Petri net model in [2]; see Fig. 7 below). Ovals represent activity states and rounded rectangles represent wait states. In an activity state, some activity is busy executing whereas in a wait state, an external event is waited for, e.g. a deadline must occur, or some third party must send some information. Wait states are also used if the current parallel branch needs to synchronise with another parallel branch. An activity state is called an action state in UML [56]. The workflow starts in the black dot (the initial state) and ends at the bull's eye (the end state). A bar represents a fork (more than one outgoing edge) or a join (more than one incoming edge). A diamond represents a choice (more than one outgoing edge) or a merging of different choices (more than one incoming edge).

State nodes are linked by directed edges, that express sequence. An edge can be labelled by $e[g]/a$ where e is an event expression, g a guard expression,

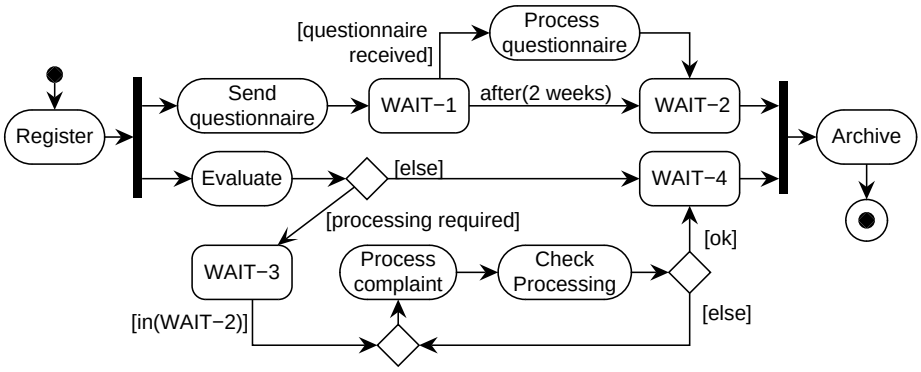

Fig. 3. Activity diagram for "Processing Complaints" workflow

and a an action expression. Each of these three components is optional. The meaning of these labels is that in order for the edge to be taken, event e must have occurred and guard g is true. When the edge is taken, the system performs action a. An edge that leaves an activity state is implicitly labelled with the completion event of the corresponding activity. We forbid that an edge that leaves an activity state has any other event expression in its label, since that would denote an interrupt, whereas an activity cannot be interrupted, since it is atomic. A special group of event expressions are the temporal event expressions, e.g. after(2 weeks) in Fig. 3 (which means that 2 weeks after state WAIT-1 is entered the corresponding edge can be taken). A guard expression can refer to variables of the activity graph. The variables of an activity graph are booleans, integers and strings. Special guard expressions are the in and else predicates. Predicate in(node name) is true if and only if the system is in state node name. Predicate else can only be used to label an edge that leaves a choice state node (represented by diamond). It abbreviates the negation of the disjunction of the guard labels of the other edges that leave the choice node. For example, the else predicate used immediately after Check processing abbreviates not ok. The only action expressions we allow are (sequences of) send event actions to specify event generation. We do not allow other action expressions in an edge label since these would change the case attributes, which we do not want, since case attributes are changed by actors, not by the WFS.

Semantics. Our semantics is based upon the following line of reasoning. We use a UML activity diagram as a workflow model. A workflow model prescribes how a workflow system should behave. Hence, a UML activity diagram prescribes how a WFS should behave. We therefore motivate and define our execution semantics in terms of workflow systems.

In our opinion, workflow systems have the following characteristics.

1. **A WFS is reactive.** A *reactive* system runs in parallel with its environment and responds or reacts to input events by creating certain desirable effects in the environment [38,58]. For a WFS, characteristic input events are activity termination events, in Fig. 3 for example that the Register activity terminates. And characteristic desirable effects for a WFS are the enabling of new activity instances.
2. **A WFS has coordination functionality.** A WFS does not execute the activities themselves, but it merely coordinates the execution of the activities by the actors (people or machines). For example, in Fig. 3 the WFS does not register the complaint itself, but merely tells the relevant actors that one of them can start registering the complaint. Case attributes are only changed in activities by actors, not by the WFS.

Each semantics is a mapping of a syntactic domain into a semantic domain. The semantic domain we use in this paper is that of a run (to be precise, a set of runs). A *run* (or a trace) is a sequence of states connected by state changes. We assume that state changes are instantaneous. So time can only elapse in a state.

Since a run is a possible behaviour of a WFS, states of the run are states of the WFS. Components of a state of a run are:

- the state of the case (i.e, which states in the activity diagram are active, possibly multiple times),
- the queue of input events of the WFS,
- the case attributes and their values, and
- the scheduled time-outs and the value of the global clock.

A queue of input events is needed because of the first characteristic: a WFS is a reactive system. In a reactive system state changes are caused by input events. This means that the WFS must have some interface with the environment to observe the input events. We therefore use an input queue in which events are kept. The case attributes are needed to evaluate the guard conditions on the edges, i.e., they are only used for routing the case.

The second WFS characteristic, coordination, has several implications. First, activities are done by the environment in states of the WFS, i.e, during an activity state the WFS waits for an activity to complete (see Section 2). Second, an activity is specified declaratively, in particular its postcondition. An imperative specification would imply that the WFS does the activity. But the outcome of an activity is not computed by the WFS. Third, in a reaction case attributes are not changed. Instead, changing (updating) of case attributes is done by the environment. Also, the WFS does not maintain the case attributes, this is done by the environment (database). But the WFS must ensure that no two interfering activities are active simultaneously. Two activities interfere with each other if they update the same case attribute. The WFS ensures non-interference by not routing to a state in which some activities interfere with each other.

Within this general picture, still a wide variety of semantics for activity diagrams can be chosen. In previous work, we have defined two semantics. The first

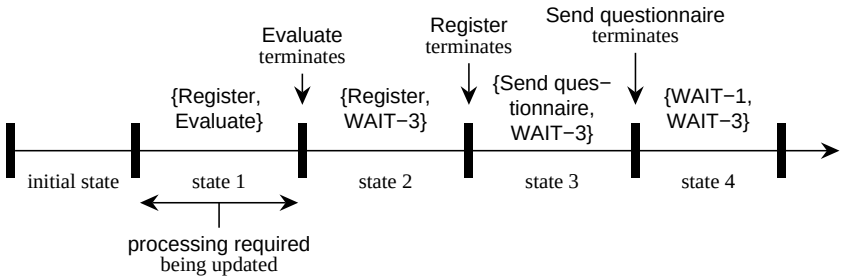

Fig. 4. Example run in requirements-level semantics

one is a *requirements-level semantics* [27] that is based upon the STATEMATE semantics of statecharts [37]. In the requirements-level semantics, the WFS is considered as a black box. In specifying requirements for the WFS, we are interested in qualitative requirements (what should be done), but not in quantitative requirements (how well it should be done), e.g. how fast a response is. We therefore abstract away from internal implementation details of the WFS. The best way to do this is to adopt the perfect synchrony hypothesis [11]. For a WFS, this hypothesis states that the WFS starts reacting to events immediately when it receives them, and also that the WFS reacts infinitely fast to these events. In a reaction, therefore, the whole input queue is read and the case is immediately routed, i.e., the state of the case is updated, and the input queue is reset again. Note that in this semantics the queue is actually a set of input events. Figure 4 shows an example run of the activity diagram in Fig. 3 under this semantics. In each state, the set of busy activities is shown.

In the second semantics [28] the perfect synchrony hypothesis is dropped. So a reaction of the WFS takes time, and input events are not immediately reacted to. We call this semantics the *implementation-level semantics*. This semantics stays close to the informal UML definition of state machines [56] (underlying UML statecharts) and the architecture of workflow systems [14,45,60,34]. In the implementation-level semantics, the WFS is considered as a white box, consisting of the components shown in Fig. 2. The Router component is responsible for producing the desired reaction: routing the case to the new state, enabling some new activity instances to start. The Router component, however, has limited capacity. It processes one event at a time (rather than arbitrarily many as in the requirements-level semantics) and it takes time to process an event (whereas in the requirements-level semantics the WFS is infinitely fast). When the Router starts routing, it picks some input event from the queue. When it finishes routing the case, it updates the state of the case, it enables some new activities to start, it schedules some new timeouts and it removes some scheduled timeouts, because they have become irrelevant in the new state of the case. Next, the Router starts processing the next event from the queue. Since the next input events might have arrived while the Router was busy with a reaction, the content of

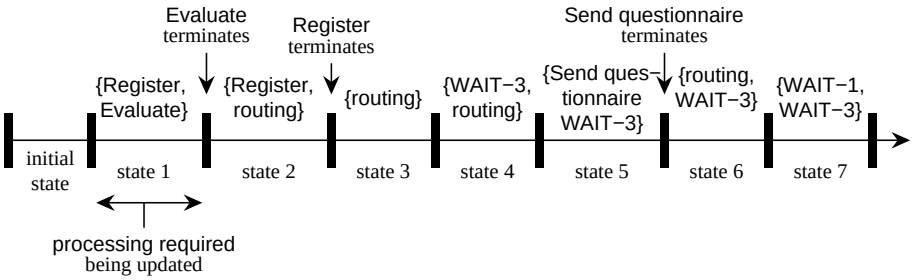

Fig. 5. Example run in implementation-level semantics

the queue might have changed during routing (whereas this is is impossible in the requirements-level semantics, since there routing is instantaneous). Figure 5 shows an example run of the activity diagram in Fig. 3 under this semantics. In this figure, the term routing denotes that the Router is busy.

Evaluating the two semantics, the requirements-level semantics is easy to analyse, but somewhat abstract, whereas the implementation-level semantics is more concrete, but more difficult to analyse, both for a workflow designer as for a verification tool. The reason for this is that in the implementation-level semantics there is a delay between the occurrence of an event and the subsequent reaction of the WFS to that event occurrence, whereas in the requirements-level semantics, there is no such delay. For example, the run in Fig. 5 is harder to match with the activity diagram than the run in Fig. 4. Moreover, the implementation-level semantics is more difficult to analyse for a verification tool, because there will be more states due to the delay in response to event occurrences.

In future work, we intend to focus on analysis of functional (logical) properties of workflow models, for example the absence of deadlock. For such properties, it does not matter whether the requirements-level semantics or implementation-level semantics is chosen: if a workflow model contains for example a deadlock in one semantics, it also will have the deadlock in the other semantics and vice versa. So we can use the requirements-level semantics for analysis of such properties with the assurance that the analysis result will also hold when the workflow model is executed under the implementation-level semantics.

Note that we do not specify how the environment behaves. In general, the exact behaviour of the environment is unknown. In our model checking semantics, we have simply assumed that the environment can behave in every possible way, i.e., chaotically, but it must respect the dependencies between value change input events (see Section 4 for more details). For analysis purposes, we assume that the environment behaves in a fair way [25].

Step semantics. The key part in both of the previous semantics is the execution of a step. A step is a collection of edges that are enabled in a certain state. By taking a step, the system reacts to events and routes the case to a new state.

In both semantics, we have adopted the same step semantics of statecharts. We here give a brief introduction to and motivation for this step semantics. More details can be found elsewhere [27,28,26].

We consider two cases. The first one is the basic case, in which action expressions on edges are not considered. Although statecharts are (in)famous for the numerous semantics invented for them, all semantics agree upon the definition of a step for the basic case that we present below. In the second case, event generation is taken into account. Adding event generation to the basic case, divides the group of statechart step semantics in two.

The basic case. In the basic case, (1) we do not consider internal event generation by the system in transitions, (2) an edge can be labelled with a single event only, and (3) we do not consider priority of transitions. Below we will add the first feature. The second feature is enforced by the UML definition [56]. The third feature is omitted from the current semantics of activity diagram, but can be added without a problem.

In the basic case, all the statechart step semantics exhibit the following three properties. This includes the most well-known ones of Harel and Naamad [37], implemented in the STATEMATE toolset as well as the different UML statechart step semantics [56], and the fixpoint semantics by Pnueli and Shalev [52].

First we list two of the three properties present in every statechart step semantics.

- **Events can occur simultaneously.** The alternative would be to assume that no two events can occur at the same time. There are two reasons for rejecting this alternative. First, although the chance of two events occurring simultaneously is rather small, it is not equal to zero. Second, the reactive system (WFS) will respond to events by inspecting the contents of the queue. If no two events can occur simultaneously, the rate at which events occur in the environment must be slower that 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. From this choice, it follows that two event occurrences are either simultaneous, or some time elapses between them. Note that this assumption is also made in our implementation-level semantics, although the Router will process only one event at a time.
- **Events live for the duration of a step only.** During execution of the system, the event queue is filled with events. The system reads the events from the event queue and reacts to them. There should however be some removal policy. If an event is not removed after it is processed, it would continue to have an effect, which is undesirable. Since the result of the event occurrences is the taking of a step, the events should be removed after this response has finished.

If the system processes events from the event queue, it reacts to these events by taking a *step*. In every statechart step semantics, a step is a bag of enabled edges that must be consistent and maximal. We explain these notions shortly.

First, the *configuration* of a system is the bag of nodes that are currently active (in Petri net terminology: a marking). An edge is *enabled* iff its sources are contained in the current configuration, its trigger events is being processed now, and its guard condition is true. A bag of enabled edges is *consistent* iff all edges can be taken simultaneously, i.e., the union of their sources is contained in the current configuration. Finally, the bag must be *maximal*, i.e., adding another enabled edge makes the resulting bag inconsistent.

The above definition of a step is a generalisation of the statechart step semantics, since in statechart the configuration is always a set, rather than a bag, and consequently, steps in a statechart are always sets, rather than bags.

In STATEMATE and in UML there is in addition a priority constraint, stating that edges with higher priority should be added first to a step. The precise definition of when an edge has priority over another one differs [42].

We now explain the third property of basic statechart steps in the basic case.

- **Steps are maximal.** Not imposing this constraint would imply that 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 not cause all their effects, although, according to the workflow model, they *should* have these effect (namely all enabled edges should be taken). In other words, then the WFS would not react fully to these input events. That is why we require that a step be maximal.

Note that the maximality constraint is motivated in terms of reactive systems. For active systems, such a motivation would no longer hold, since then the system can decide itself what to do and does not have to do as much as possible. This gives an explanation for the fact that in Petri nets the maximality constraint is usually not adopted.

We emphasise that statechart steps in the STATEMATE semantics, UML semantics and the fixpoint semantics all share these three properties.

Event generation. Next, we extend the basic case by allowing for *event generation* by the system. An event can be generated by taking an edge. Figure 6 gives an example (send actions follow the slash). Some of the edges have an identifier for ease of reference. The label on the edge t1 states that if the edge is taken (because e occurs and WAIT-1 is active), then event f is generated. Please note that communication by means of event-generation is always asynchronously, not synchronously (rendez-vous or hand-shaking).

There are two different ways of interpreting event generation. The first one is to let the generated events have an effect in the current step (chosen in the fixpoint semantics [52]), the second one is to let the generated events have an effect in the next step (chosen in the STATEMATE semantics [37]). To illustrate the differences between these two options, suppose in Fig. 6 the current configuration is [WAIT-1,WAIT-2] and event e occurs. If edge t1 is taken, then according to the fixpoint semantics event f is immediately available and consequently edge t3 can be taken simultaneously with edge t1. Whereas in the STATEMATE semantics,

event f can only be sensed *after* the step in which it is generated is taken, so *after* edge t1 is taken. Consequently, if the current configuration is {WAIT-1,WAIT-2}, and event e occurs, in the fixpoint semantics either step {t1,t2} or {t1,t3} is taken, but in the STATEMATE semantics, step {t1,t2} is taken. The step {t1,t3} is counter intuitive here, since it seems that event e is ignored in node WAIT-2. So there are circumstances in which the fixpoint semantics computes a counter intuitive step (this was first pointed out by Leveson et al. [43] using a similar example, but they mistakenly attribute the fixpoint semantics to STATEMATE). That is why in practice the STATEMATE approach is taken, even in the UML, where events are called signals. (We do not have operation calls in our activity diagram semantics.) We have adopted the STATEMATE interpretation for event generation as well, since it is also adopted by UML. As an aside, note in this interpretation too, there are anomalies. One may for example get infinite loops in which events are generated for ever, because some events cause each other to occur [43].

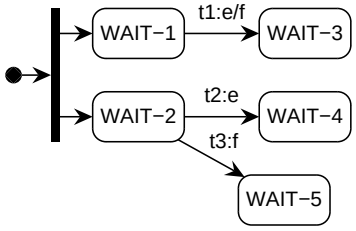

Fig. 6. Event generation

The state of the practice. We do not know of any commercial WFMS that allows for the specification of workflow models using UML activity diagrams. But few of the current commercial workflow systems offer some support for modelling events [13]. We therefore expect that the constructs of our semantics related to events will be hard to express in workflow models of existing commercial WFMSs. On the other hand, our event broadcast semantics, in which one event can trigger more one edge, is similar to the publish-subscribe notification mechanism used in middleware application and recently adopted in the industry standard for workflow interoperability [50], defined by OMG and WFMC. Also, in active database systems [57] an execution semantics for rules is adopted that is similar to the semantics for edges that we use. In particular, in active databases a generated event has an effect in the next step, not in the current step. So in active databases also the STATEMATE interpretation is chosen (although the link with statecharts is not made by for example Widom and Ceri [57]). Also, the possibility of nontermination of the rule processing algorithm due to rules that trigger each other, is a well known feature of active databases [57].

Recently, UML activity diagrams have been proposed to model e-business services in e-business standards like ebXML [21]. We expect that process management tools that support e-business services will use UML activity diagrams. In for example ebXML, the event features of activity diagrams are used quite extensively: events are the standard means of communication between different business partners. Events are also used quite extensively in business modelling, especially in UML-based approaches, for example [24]. In academia, several WFMS research prototypes use event-based workflow models (e.g. [13,33,35,59]), often inspired by active databases [57].

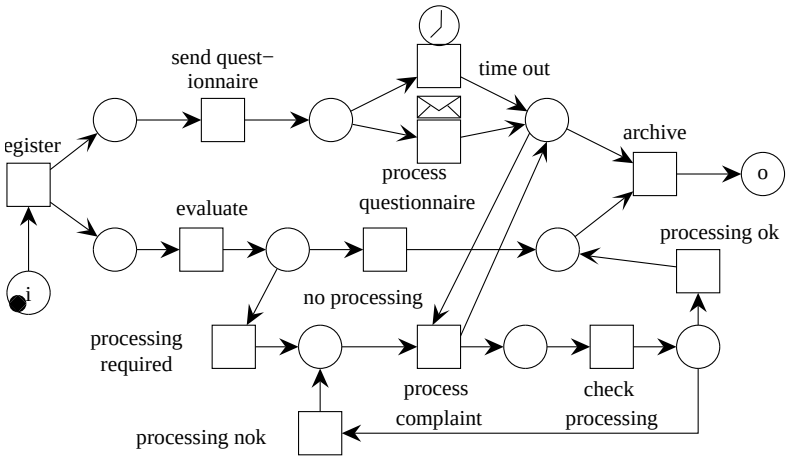

Fig. 7. Petri net for "Processing Complaints" workflow [2]

4 Modelling Workflows with Petri Nets

We now investigate how several – what we think are – important aspects of workflow models are modelled in Petri nets. We will compare the Petri net workflow models with our two semantics of UML activity diagrams. We take our requirements-level semantics as point of comparison, since it most resembles the Petri net semantics. At the end of this Section, we will discuss how the implementation-level semantics can be modelled in Petri nets. In order to make a fair comparison we assume that a Petri net models a WFS too.

Remark on terminology: from now on, we will use the standard Petri net terminology of place (corresponds to state node), transition (corresponds to edge), and marking (corresponds to configuration). By "step" we mean a statechart step, unless stated otherwise.

4.1 Modelling Events

Several researchers that use Petri nets for workflow modelling have recognised the importance of input events for workflow modelling ([2,41]), even though using the a different name: 'trigger'. Figure 7, taken from Van der Aalst [2], presents a typically example of the use of input events in a Petri net. (This Petri net models the same workflow as the activity diagram in Fig. 3.) The envelope and the clock denote external and temporal trigger events respectively.

Unfortunately, although the importance of input events is recognised, hardly ever a semantics is given for them. Van der Aalst [3] gives an interesting motivation for abstracting from events for analysis purposes, that we will discuss in Section 4.6. But first we study two approaches to model events in ordinary Petri nets and compare both approaches with our semantics of input events.

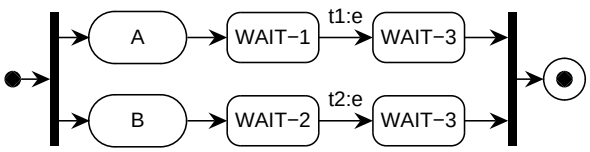

Fig. 8. Event broadcasting

Event as token. For each input event a place is defined. The place represents a kind of interface with the environment. If the interface place is filled with a token, the input event occurs, otherwise it does not occur. The interface place is connected with all transitions that are triggered by the input event.

This is the approach taken in Trigger Modelling [41]; it is also suggested as an appropriate semantics for trigger events in Workflow Nets [2]. In these approaches, the environment is not specified, but the suggestion is made that the environment fills the interface places spontaneously, but no formal semantics is presented. Open nets [10] gives a formal semantics for nets with interface places, which could be used for Trigger Modelling and Workflow Nets.

One important difference of the event-as-token approach with our semantics of events is that in the event-as-token approach one event occurrence triggers at most one transition whereas in our semantics one event can trigger more than one transition (edge). This is because we have event broadcasting in our semantics, rather than point to point communication. For example, in Fig. 6 one occurrence of event e can trigger the two edges `t1` and `t2` simultaneously. Since in the standard Petri net semantics, firing a transition implies that its input tokens are consumed, in the standard Petri net semantics only one transition can fire because of one event occurrence.

One might wonder whether event broadcasting is desirable. In other words, isn't the standard Petri net interpretation of consuming events, so having an event trigger at most one transition, better? We think event broadcasting is desirable for the following reasons. First, if an event would trigger a single transition only, the event would not have all the effect that is specified in the workflow model. For example, a cancel event that stops a workflow would be awkward to model. In the event-as-token approach, a cancel event could only stop one parallel branch, whereas in our semantics an event is global for the whole workflow and there a cancel event can stop the whole workflow. To cancel a workflow in the event-as-token approach, for every parallel branch a separate cancel event needs to be generated.

Second, the broadcast mechanism is used quite extensively in the field of workflow systems. Several non-Petri net based WFMS prototypes [13,33,35,59]) also use a broadcast semantics in their workflow models. The industry standard for workflow interoperability [50], defined by OMG and WFMC, uses a so-called publish-subscribe notification mechanism, which is similar to our broadcast semantics. An exception are XML and EDI based workflow specifications,

which currently only use point-to-point communication between business partners. However, some of these approaches [21] will adopt publish-subscribe notification in the near future. Also, these approaches do not specify what communication mechanism are used within an organisation, since it falls outside the scope of these frameworks. So even in these approaches, a broadcast mechanism can be used for intra-organisational communication.

Third, we observe that our broadcast semantics is equivalent to a point-to-point semantics if all the used event names in the activity diagram are unique. But, as we will explain next, it is not possible to fully capture the broadcast semantics with point-to-point communication, since the exact addressee is not always known at design time and may depend upon the current state of the case.

There are several ways to simulate the effect of event broadcasting in Petri nets. The most obvious one is to use transition fusion and glue the edges with the same event label together. Although this would work for the example in Fig. 6, this is only a partial solution, for two reasons. The first one is that it depends upon the current configuration (marking) of the activity diagram whether or not two edges are taken simultaneously. For example, in Fig. 8 the two edges are only taken simultaneously if the current configuration is [WAIT-1,WAIT-2]. Otherwise, if for example the configuration is [WAIT-1,B] and e occurs, than only t1 is taken and configuration [WAIT-3,B] is reached. So, only at run-time it is known which edges need to be fused together, whereas transition fusing is applied at design time. Second, applying transition fusion at design time does not solve this problem, since the original edges cannot be left out. For example, if in Fig. 8 edges t1 and t2 are fused together into t12, then edges t1 and t2 must remain in the model, since it possible that either one of them is taken separately from the other. Consequently, if the current configuration is [WAIT-1,WAIT-2] and event e occurs, it still might be possible that only say t1 is taken, and not the fused edge t12.

Another possible way to simulate event broadcasting is to fill the interface place with as many tokens as needed to prevent that a transition cannot fire because of a lack of tokens. But the exact number of tokens that is needed is not known beforehand, since the number of transitions to be fired depends upon the current WFS state. Consequently, a lot of spare tokens would have to be introduced. This blurs the difference between two occurrences of the same event at different times and two copies of the same event occurrence. Although this could be resolved by time stamping tokens, the resulting semantics would be overly complex and more involved than the statechart step semantics.

A better alternative is to model the control flow between an interface place and a transition that it triggers as a read arc [47], also known as context relation. A read arc from a place to a transition means that although a token must be present in the place to let the transition fire, this token is not consumed. (A read arc from a transition to a place is impossible.) Technically, a flow relation is added to the usual Petri net semantics that specifies the read arcs [47].

But even using read arcs, the event semantics we give to value change events cannot be adequately modelled. A value change event is the event that a boolean

condition on some variable x becomes true, e.g. $[x > 10]$. In our semantics, if there are two value change events referring to the same variable, e.g. $[x > 10]$ and $[x > 15]$ then there *might* be a dependency between them, e.g. if $[x > 15]$ occurs then $[x > 10]$ *might* also occur, but not necessarily. For example, if x changes from 11 to 16, then $[x > 15]$ occurs but $[x > 10]$ does not (since x already was greater than 10). But if x changes from 9 to 16, both events do occur simultaneously. This cannot be modelled faithfully in the event-as-token approach (even a dependency relation between interface places, stating something like "if that interface place is filled, this one must be filled as well" does not solve this problem).

A final 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 triggered by e.

Event as transition. In Petri nets, one can simulate an event by labelling a transition with the event name and interpret the firing of the transition as the event occurrence. By specifying synchronisation constraints [15,29] between the event transition and the system transitions, it can be specified that an event occurrence triggers a system transition. Note, however, that then the environment is being modelled explicitly, rather than implicitly as in the event-as-token approach. In other words, the whole Petri net is now a model of both the environment and the WFS, rather than of the WFS only.

One advantage of this semantics is that it is very easy to specify that one event occurrence can trigger more than one system transition, since the synchronisation constraint is specified as just a relation between transitions.

But, as in the event-as-token approach, the dependency between value change events cannot be modelled faithfully, because the presence of the dependency depends upon the previous value of the corresponding variable.

Conclusion. We conclude that in both the event-as-token and the event-as-transition approach, we cannot model our full event semantics, since the dependency between value change events cannot be modelled completely. But both the combination of open nets with read arcs, and nets with synchronisation between transition are certainly steps in the right direction. In the next subsection, we will study how well the statechart step semantics can be modelled using these two approaches.

4.2 Modelling Steps

In the previous subsection we identified the two ways, open nets with read arcs, and nets with synchronisation constraints between transitions, that come closest to our event semantics (but nevertheless they are still different from it). We now study whether and how well the statechart step semantics can be modelled in these approaches. We focus both on the basic statechart step semantics and the event generation semantics. For each approach, we study whether the three properties of the basic statechart step semantics are met or not. And if they are not met, we study whether they *can* be met.

Event as token. If we take the Petri net step semantics, it is no problem in this approach to model that events can occur simultaneously. But in the event-as-token approach, it is difficult to specify that events live for the duration of one step only, without changing the semantics of open nets at this point. The removal of an event occurrence has to be modelled by a separate transition that removes the token from the interface place. But the sequence 'event occurrence-system reaction-event removal' which is key part of the basic statechart step semantics, is not part of the standard Petri net semantics. It seems to us that it is impossible to model this sequence using standard Petri net semantics, since in this semantics any sequence of transitions, obeying the firing rules, is allowed. So, it could be possible that under the standard Petri semantics an event lives longer than a step.

Recently, a new Petri net variant, called zero safe nets [12], has been proposed that seems a good starting point for modelling the statechart step semantics. In zero safe nets, some places, called zero places, represent unobservable system states. A marking in which one or more zero places are filled is unstable, otherwise it is stable. During execution, the system moves from one stable marking (in which zero places are not filled) to another stable marking via a sequence of unstable markings. By modelling event places as zero places, the statechart step semantics can be simulated to some extent. Still there is a difference: zero safe nets have a constraint that all stable tokens present at the begin stable marking must be consumed during the sequence. For the statechart step semantics, this would mean that relevant transitions must fire immediately, which is of course not true.

Finally, the constraint that steps are maximal is not present in standard Petri net semantics. Rather, steps in the Petri net semantics can be any consistent subset of the bag of enabled transitions. Of course, the maximality constraint could be added without a problem (like incidentally done by some authors, e.g. [29]), but it does not seem very intuitive for the standard Petri net semantics. In fact, Foremniak and Starke [29] have considerably changed the standard Petri net semantics. We discuss their approach in more detail below.

Event as transition. There are several Petri net variants that have incorporated synchronisation between transitions in their models[15,8,29]. The work of Christensen and Hansen [15] introduces the concept of synchronous transitions. They focus on symmetric synchronisation. Object-oriented Petri nets [8,46] use both symmetric and asymmetric synchronisation between transitions, i.e., one transition has the initiative, the other one follows. All these references stick to the standard interleaving semantics, which differs considerably from the statechart step semantics, amongst others because the maximality constraint is not required.

Finally, in signal-event nets [36,29] the standard Petri net step semantics is abandonded in favour of a semantics in which also a maximality constraint is adopted. Signal-event nets are introduced by Hanisch and Lüder [36] in order to model discrete event systems. To model a discrete event system, both the behaviour of an uncontrolled plant and of a controller that guides the behaviour

of the plant is modelled. Hanisch and Lüder argue that discrete event systems cannot be faithfully modelled using ordinary Petri nets. Discrete event systems are an excellent example of reactive systems: the controller must react to the behaviour of the plant and it does this in order to maintain the plant in a desired state. It is therefore interesting to note the similarities (and differences) between the execution semantics of signal-event nets and that of statecharts. Foremniak and Starke [29] introduce an execution semantics for signal-event nets. The key part of the execution of a signal-event net is a step. Before we discuss their definition of a step in more detail, we fix some terminology [29]. A transition is *forced* it is triggered by another transition; otherwise it is *spontaneous*. (So in standard Petri nets, every transition is spontaneous.) A transition t can be forced by more than one transition. There are two options in that case: either all trigger transitions must occur simultaneously to trigger t (AND), or only one trigger transition has to occur in order to trigger t (XOR). We only consider the XOR interpretation here.

A set s of transitions is *signal complete* iff

i if s only contains spontaneous transitions, it is signal complete.
ii if s is signal complete, $t \notin s$ is forced and t is triggered by a transition $t' \in s$, then $s \cup \{t\}$ is signal complete.

A step s must satisfy the following constraints:

1. s contains transitions that are fired spontaneously, i.e., without being triggered by another transition,
2. the input places and input conditions (for read arcs) contain sufficient tokens for all transitions in the step to fire,
3. s is signal-complete,
4. for every non-spontaneous transition t' that is not in s, set $s \cup \{t'\}$ does not satisfy 1-3.

We can easily see the correspondence with our semantics: spontaneous transitions are transitions in the environments, representing events, whereas forced transitions are transitions in the workflow model (so done by the WFS). Constraint 1 states that every step must be triggered by at least one event. Constraint 2 states that all the transitions in the step must be enabled and consistent. Constraint 3 says that a transition that is triggered by an input event e can only be part of the step if e occurs. Constraint 4 states that the step be maximal. It is not difficult to see, that these constraints are indeed equivalent to the constraints we discussed in Section 3 for the basic statechart step semantics.

But, this definition differs with our semantics w.r.t. the generation of events during a step, since it assumes that events generated by the system are sensed immediately in the same step (as in the statechart fixpoint semantics [52]), whereas we assume that they are sensed after the current step has been taken (as in STATEMATE [37] and UML [56]). To illustrate this, we translate the activity diagram in Fig. 6, shown in the left-hand side of Fig. 9, into a signal-event net, shown as the right-hand side of Fig. 9. The interrupt arcs represent triggering;

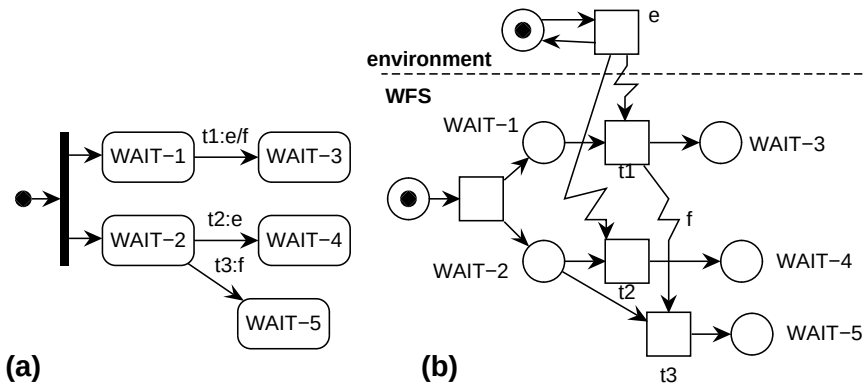

Fig. 9. Event generation modelled in signal-event nets

the triggered (forced) transition is pointed at. Suppose the current marking of the signal-event net is [WAIT-1,WAIT-2] and transition e occurs. Then both [e,t1,t2] as [e,t1,t3] are valid steps, according to the constraints listed above. This is similar to the behaviour of the corresponding activity diagram (statechart) under the statechart fixpoint semantics of Pnueli and Shalev [52], as we explained in Section 3. But in our semantics [27,28], only [t1,t2] would be possible. As explained in Section 3, we regard the fixpoint semantics (and thus the signal-event step semantics) as counter intuitive here, since it seems that e is ignored in state node WAIT-2 if step [t1,t3] is taken.

It is easy to show that the signal-event net execution semantics is a strict subset of the fixpoint statechart semantics (strict because in the statechart variant on which the fixpoint semantics is defined, an edge can be labelled with a negative event ($\neg e$), which is true iff the event does not occur. Negative events cannot be defined in signal-event nets).

A more intricate example is presented in Fig. 10. The predicate in(x) that is used in the activity diagram, evaluates to *true* iff node x is contained in the current configuration. It can be translated into a Petri net construct using read arcs and inhibitor arcs. Inhibitor arcs are necessary to model not in(x). In Fig. 10, read arcs are lines, so do not have an arrow; inhibitor arcs are lines with a circle at the transition end. If the current configuration is [A,WAIT-1] and at the same time activity A terminates and event e occurs, then in our semantics a sequence of steps is taken such that finally configuration [final,B] is reached. But in the corresponding signal-event net, configuration [final,WAIT-2] is reached and event *done* is not responded to and is lost! Consequently, the final configuration will never be reached in that case. This is clearly undesirable. We therefore prefer our semantics of event generation.

Of course, now the question arises whether our semantics of event generation can be simulated in signal-event nets. We think that this is impossible, since in

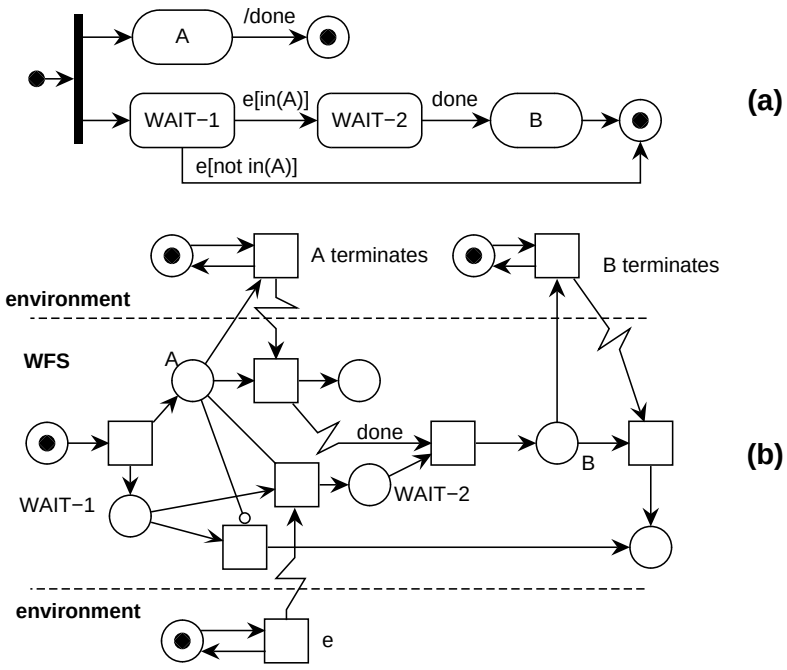

Fig. 10. Activity diagram and a similar signal-event net

both our semantics [27,28] the input set acts as a kind of registry in which the events that are generated during a step are stored. This can only be simulated by treating events as tokens; if events are treated as transitions, events get lost after the step in which the events are generated completes. But above, we discussed the inadequacy of the event-as-token approach to model our step semantics. We come back to this issue in the conclusion of this subsection below.

Another point is that in general, statecharts have a priority constraint to deal with certain forms of nondeterminism. These are not present in the definition signal-event nets. Other differences between activity diagrams and signal-event nets are that (1) steps in signal-event nets are *sets* of transitions, rather than bags (but we could not find a compelling reason in Foremniak and Starke [29] why this is the case; probably the extension to bags is easily made), and (2) the environment must be modelled explicitly in signal-event nets, but not in activity diagrams.

Conclusion. Both in the event-as-token approach as in the event-as-transition approach, the statechart step semantics we have adopted cannot be modelled. However, in the event-as-transition approach, the signal-event net semantics resembles the statechart step semantics we use closely. But the signal-event semantics has a fixpoint semantics of generated events, whereas we have not. Our semantics of event-generation can only be modelled using the event-as-token ap-

proach. It might therefore be worthwhile to try to incorporate the concepts used in signal-event nets into the event-as-token approach. Especially the concept of a forced transition seems promising. This concepts seems to be present in zero safe nets as well.

4.3 Modelling Data

The standard way to incorporate data in Petri nets is to use coloured tokens [40]. Coloured tokens are tokens that have attribute values. These attributes values are modified in/by transitions. Another way is to interpret places as predicates [31]. But then instances of the predicates can be seen as tokens that can change value when a transition consumes them. So, in both approaches, tokens carry data.

Therefore, the straightforward way to model case attributes in Petri nets is to attach these attributes to tokens. But attaching case attributes tokens suffers from the following problems.

Who updates case attributes. If case attributes are updated in some transition, than this transition cannot be part of the workflow model, because the WFS who executes the workflow model does not update case attributes, it only routes the case. (See Section 3). In other words, the environment (the actor) must be specified explicitly by a transition in order to let the case attributes change value.

Data integrity. Several tokens may represent the same case attributes. Ideally, this situation should be prohibited, since an attribute may then inconsistently have several different values (i.e., the different tokens may assign different values to the same attribute). In terms of transaction theory, the isolation property fails to hold, since activities that update the tokens are not isolated from the other executing activities.

One possible solution is to represent each case attribute by a single coloured token. Then each transition that reads or writes the attributes must have this token as input and outputs the token when it finishes. Although the isolation property is then ensured, in standard Petri nets two read activities cannot be simultaneously active, since both consume the same token. That is not what we want, because the concurrency of the WFS is then reduced. In addition, the resulting net would look like ravioli if there are many case attributes.

To circumvent this, read arcs [47] can be used for read access. Interestingly, apparently read arcs have been proposed just to solve this problem of simultaneously access to shared data [30]. But unfortunately, read arcs do not solve the problem satisfactorily. To illustrate this, consider the Petri net with read arcs in Fig. 11. Data item x is updated and read by activity A and read only by B. In this net, although x cannot be updated and read simultaneously, it is possible that A reads a value of x that is subsequently changed by B. So, A and B are not isolated from each other (viewing both activities A and B as separate transactions.) Therefore, this solution does not satisfy our needs. (De Francesco et

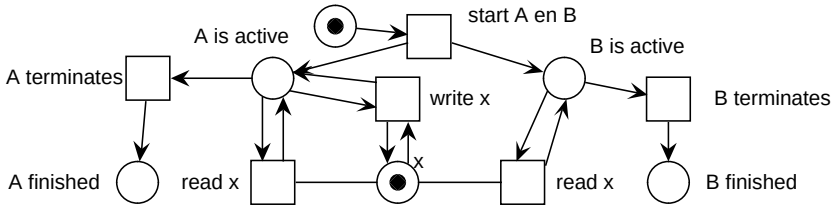

Fig. 11. Example of concurrent access to shared data. Activities A and B both access data item x

al. [30] do not address this issue; they only consider the question when two Petri net executions are view equivalent.)

In fact, in our semantics [27] we have ensured that if two activities are conflicting, that is, one of them writes a case attributes that the other one reads or writes, then they cannot be active simultaneously. In the computation of a step, we have put the extra constraint that by taking the step a configuration is reached that has no conflicting activities. This conflict relation can of course be specified in the control flow as well, using for example a mutex place for each pair of conflicting activities. The mutex place acts as a kind of semaphore: the activity that can consume the token in the mutex place may be active and change the data item it likes and when it terminates it puts a token in the mutex place. A solution using mutex places would, however, clutter the workflow model with a lot of arrows, and again, we have a ravioli model, that is even more unreadable and incomprehensible than the workflow model presented in Fig. 11.

Conclusion. We conclude that data can be modelled in Petri nets using read arcs, mutex places, and an explicit representation of the environment in the form of actors that update the case attributes. But, the resulting net is overly complex, unreadable and uncomprehensible. We think that a solution using local variables (used in Petri nets modelling flowcharts [32]) is more simple and elegant and therefore preferable.

4.4 Modelling Activities

In a Petri net, there are two options to model an activity: as a transition or as a place. Almost every Petri net workflow model seems to take the first option, whereas if an activity diagram is viewed as a Petri net, the second option seems to be taken. We discuss the advantages and disadvantages of each option.

Activity is transition. In every Petri net workflow model that we know of, this interpretation is adopted, probably because of the intuition that an activity is something which changes the state of the case (the state is assumed to be modelled by the input tokens). There are, however, some mismatches between the properties of an activity and the properties of a transition. First, a transition

takes no time to execute, whereas an activity does. There are two ways to solve this problem. The first solution is to decompose the transition into a "begin activity transition" and "end activity" transition that are connected by a place representing "activity busy executing". This solution results in a Petri net that is quite similar to an activity diagram. Then the execution of an activity is actually represented by a place. This approach is taken by for example Van der Aalst, Van Hee and Houben [6] and Desel and Erwin [19]. See the next item below for a discussion of this approach.

The second solution is to use timed or stochastic Petri nets, in which a transition can have a duration. In most timed and stochastic Petri net variants a transition still fires instantaneously, but it takes time before a transition is enabled. The transition in that case actually represents the starting or ending of an activity rather than the complete execution of the activity. This is not harmful for analysis purposes, but it gives a slightly awkward model of WFS reality. Analysis of timed and stochastic Petri nets is far more complex and involved than analysis of simple low-level nets.

However, our main objection against modelling an activity as a transition is the following. In Petri nets, a transition is executed by the system that the Petri net models. Hence, if a transition models an activity, this implies that the WFS does the activity. This approach violates the WFS characteristic that an activity is performed by an actor in the environment of the WFS, not by the WFS itself. *And it is this characteristic that creates the need for reactivity in a WFS.* By contrast, in our semantics the WFS does not do activities; it merely routes cases. In Petri nets that model activities as transitions, the routing is not modelled at all. Therefore, such Petri nets do not model a WFS.

As an aside, note that in some variant of Workflow Nets [5], some transitions can be labelled with a silent action that is not observable for the environment. Van der Aalst [4] suggests to use the silent step to model routing transitions. Transitions labelled with an observable action then represent workflow tasks. However, in that semantics, the silent action can be abstracted away from sometimes. For example, a sequential workflow model with two tasks a and b and a routing transition from a to b is equal to a model in which a is directly followed by b. It is unclear how this abstraction can be related to the execution of real workflow models: a WFS always routes a case after an activity completes. In our view, routing cannot be abstracted away from.

Of course, one could model the environment also in the Petri net workflow model, and let the activity be performed by the environment part of the Petri net model. But then the relationship with the corresponding part of the workflow is unclear, i.e., what should the WFS do while the environment is busy performing some activity?

Activity is place. To the best of our knowledge, this interpretation is never chosen in Petri nets. Most people modelling a workflow in Petri nets probably would find this interpretation counter-intuitive since (as they argue) during an activity the case is changed, whereas a place is static (the local part of the case is not changed). We disagree, however, with this argument, since for a WFS

an activity state *does* represent something static, namely the WFS waits for an actor to complete the activity. The only dynamic behaviour of the WFS is when some activity completes and the case must be routed to a new state.

Nevertheless, the Petri net people who find this interpretation counter-intuitive are right to some degree. Whether we represent activities as places or as transitions, in Petri nets case attributes can only be changed in transitions, not in places. This corresponds to the fact that Petri nets model closed, active systems, in which the environment, i.e., that which it outside the Petri net, does not play any role. Any model of an open, reactive system, on the other hand, does allow for a change of case attributes during a state (place), namely if the change is initiated by the environment! (These changes are implicitly modelled and not explicitly represented by an edge in the diagram.) And this is exactly what happens during an activity: the environment (i.e., an actor using an application) updates case attributes, whereas the WFS waits for the activity to terminate. Consequently, to model change of case attributes in a Petri net, we must model the environment explicitly in the Petri net as well.

Conclusion. As stated above, in most Petri nets workflow models the first option is adopted. The only motivation that is given for this interpretation is that this is "straightforward". We think that the real, underlying motivation is based upon the following properties of Petri nets: (1) a transition represents some change by the system, whereas a place represents a static condition on the Petri net model, and (2) a Petri net model can only change state by firing transitions. The two properties imply that all changes are caused by behaviour of the system itself. In other words, changes cannot occur due to the environment of the system. Consequently, any modelling language having these properties cannot faithfully model open, reactive systems; instead, they are more suitable for modelling closed, active systems. Petri nets are useful for example to represent (scarce) resource usage, e.g. the allocation of actors to activities.

4.5 Modelling the Implementation-Level Semantics

In the implementation-level semantics, only a single event can be processed at at a time. Therefore, a queue is needed to store events that occur while the Router is busy processing some event. We now discuss whether this can be simulated using the event-as-token and event-as-transition approaches.

In the event-as-token approach, a queue can be modelled straightforwardly by switching to Petri nets with integers (counters) as is done in the FunSoft approach [23]. And a special place can be introduced to store the event that is currently being processed by the Router. But still, since in the event-as-token approach the statechart step semantics we use, cannot be simulated very well, the implementation-level semantics cannot be simulated very well either.

In the event-as-transition approach (signal-event nets), queues cannot be modelled, since the effect of an event is lost after the step in which it occurs is completed. So, if in the environment an event occurs (a spontaneous transition fires), the WFS must react immediately, since otherwise the event will be lost. So,

in the event-as-transition approach, the implementation-level semantics cannot be simulated at all.

We conclude that the implementation-level semantics cannot be modelled satisfactorily using Petri nets.

4.6 Petri Nets for Workflow Modelling

We conclude this Section by discussing the Petri net models we found in literature that are used to model and analyse workflows. Van der Aalst, Van Hee and Houben [6] use high-level nets to model and analyse Petri net based workflow models that also model resources. Van der Aalst [3] uses Workflow nets, low-level Petri nets with a single start and a single end place, to verify proper termination of a workflow model. Although Van der Aalst recognises the need for modelling input events, he abstracts away from them for analysis purposes for two reasons [3]. His first argument is that the environment cannot be modelled completely; from the point of view of the WFS it behaves nondeterministically. This is best modelled, he says, by leaving the events out. His second argument is that if an abstracted workflow is correct, the concrete one will also be. But as we showed above both arguments fail to hold if there is a dependency between different events in the same workflow model and if one event can trigger more than one transition. In these cases, abstracting from events will lead to different behaviour in the abstract model, when compared to the concrete model. Consequently, the verification results obtained for the abstract net might not be reliable anymore.

FunSoft nets [18] are high-level nets for software process modelling, but they can also be used for workflow modelling. Their semantics is defined in terms of Predicate/Transition nets [31]. FunSoft nets focus on the flow of resources (objects), like business documents, through an organisation and do not focus on modelling events. Some shorthands are defined to model for example FIFO queues. Several analysis techniques, including verification have been developed for FunSoft nets [18].

INCOME/WF [49] is a a workflow management system based on high-level Petri nets where the tokens are nested relations. Nested relations are introduced to increase the concurrency of the net: the actual transitions are defined on the basic elements of the relation, not on the relation itself. This means that still for the basic elements, no concurrency exists, since the standard Petri net firing rule is employed, in which a transition consumes all tokens it reads.

Information/Control Nets [22] are a high-level Petri net variant for workflow modelling. The focus is on the modelling of resources, like documents, not on the modelling of events. The standard Petri net semantics for high-level nets is used.

Milano [9] is a research prototype to investigate the issue of flexible workflow models. The Petri nets that are used do not contain events, loops, data, real-time. Moreover, these nets must be safe.

It is interesting to notice, from this brief overview, that most Petri net workflow models provide little support for modelling events. And if a notation for

events is suggested, no formal semantics for them is given. Most approaches interpret tokens as resources that are being used by activities in transitions to deliver a requested service for a customer. However, the assumption seems to be implicitly made that resources are *scarce*, since no two transitions can consume the same token simultaneously. This assumption is questionable: certainly, some resources are scarce, but also a wide variety of resources, especially information carriers, are not. (None of these Petri net variants uses read arcs.)

5 Discussion and Conclusion

From our comparison of our semantics with Petri net semantics, we draw the following conclusions. First, Petri nets model closed systems. All changes in Petri nets occur because of the firing of some transitions in the net that represent activity of some part of the system itself, rather than some activity in the system's environment. Second, standard Petri nets model active systems, rather than reactive ones. A transition is enabled if its input places are filled. Also, an enabled transition does not have to fire immediately. Our semantics is reactive. An edge in an activity diagram is enabled if its source state nodes are in the current configuration and its trigger event occurs in the environment. And an enabled edge *must* fire immediately. That is why we impose a maximality constraint on steps in our semantics. This constraint is lacking for standard Petri nets.

In Petri nets, reactivity can be simulated to some extent by modelling the environment in the Petri net as well. This is done, for example, in a recently proposed variant of Petri nets, called signal-event nets. These nets are also motivated by the domain of reactive systems. Signal-event nets have a complex semantics that differs considerably from the standard Petri net token-game semantics (amongst others, a maximality constraint is imposed on steps). Since their semantics is so different from the token-game semantics, it is questionable whether these are Petri nets at all. We showed that signal-event nets behave similar to the fixpoint step semantics for statecharts, defined ten years earlier by Pnueli and Shalev [52]. We are convinced that it is impossible to simulate in Petri nets the STATEMATE interpretation of event generation, that is also used in UML. However, it might be worthwhile to try to incorporate the concepts of signal-event nets into Petri net variants that model events as tokens, for example open nets. The resulting Petri net variants would likely be closer to the STATEMATE interpretation of event generation than any of the currently existing Petri net variants, but we expect such variants will still be different.

Third, Petri nets in general model scarce resources, rather than unscarce ones. A transition can only fire if there are enough input tokens present, i.e., enough scarce resources are available. Using read arcs this can be circumvented, because with a read arc a token can be tested without being consumed. Thus, read arcs allow for elegant specification of concurrent access to shared data. However, if an activity is seen as a transaction, as is usually done in workflow modelling, read arcs must be combined with mutex places to enforce isolation

between activities. We prefer our own approach using local variables, since it is more simple.

Fourth, the Petri nets that came closest to the requirements-level semantics we gave to activity diagrams contained inhibitor arcs, read arcs, synchronisation between transitions, and coloured tokens with timestamps. These nets had to contain both a description of the workflow and of the environment. Roughly speaking, the net had twice as many nodes compared to the corresponding activity diagram. Such Petri nets are truly gargantuan and difficult to analyse, both for a workflow modeller and a verification tool. (For example, a lot of the analysis results for standard Petri nets do not carry over to signal-event nets [29].) And even with these Petri nets, the possible dependencies between value change events could not be modelled. Moreover, these Petri nets still stay far away from our implementation-level semantics. They do not resemble it at all.

Also, one of the acclaimed advantages of Petri nets, that there is an abundance of analysis techniques available for them, is only true for low-level nets; its applies to a lesser extent to high-level nets. As we showed, not every desirable construct can be modelled in low-level and high-level nets; read arcs, inhibitor arcs and synchronisation constraints are needed as well. But for these latter net variants, there are only very few analysis techniques available.

Of course, these conclusions are based upon our assumption that a workflow model describes the behaviour of a WFS. One could argue whether this assumption is valid. In fact, does not a Petri net workflow model describe an organisation, rather than a computerised system? But even then, the interaction between the organisation and its environment must be modelled (customers, government, suppliers,...), since the organisation is a reactive system as well. Consequently, Petri net models of organisational behaviour suffer from the same problems as Petri net models of WFS behaviour: they cannot model reactivity.

From the above, it follows that if a Petri net is used to model a reactive system, the reactivity of that system is abstracted away from. But we consider reactivity to be one of the most important aspects of workflow modelling. If reactivity is abstracted away from, then at least some justification should be given that assures that the analysis results on the Petri net model will also carry over to a reactive setting. We do not think such justification has been given yet. If no justification is given, it is unclear what the relationship is between a Petri net model and the actual execution of a similar workflow model by a WFS. We conclude that Petri nets cannot faithfully model reactive systems, unless they are extended with a syntax and semantics for reactivity

Finally, Petri net variants are sometimes used as workflow modelling language in workflow management systems, for example in Cosa [44] and MQSeries [39,45]. This does not mean, however, that these WMFSs also use the Petri net semantics! For example, in Cosa, activities are modelled as transitions. In the token game semantics of Petri nets, a transition fires instantaneously whereas in real life an activity will not be performed instantaneously. So, although the Petri net syntax is used, it is very doubtful whether the Petri net semantics is used. We think that this observation holds for other workflow management tools as well, since

most adhere to the reference model of the Workflow Management Coalition [60] that, as we do, views workflow systems as reactive systems (cf. Fig. 2) that have coordination functionality.

Acknowledgements. The authors would like to thank Wil van der Aalst and Jörg Desel for their critical comments on a previous version of this paper.

References

1. W. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management*. LNCS 1806. Springer, 2000.
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. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In Aalst et al. [1], pages 161–183.
4. W.M.P. van der Aalst. Personal communication, 2001.
5. W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theor. Comp. Sci.*, 270(1-2):125–203, 2001.
6. W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proc. 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.
7. 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, pages 18–29. Springer, 2000.
8. G. Agha, F. Decindio, and G. Rozenberg. *Concurrent Object-Oriented Programming and Petri Nets*. LNCS 2001. Springer, 2001.
9. A. Agostini and G. de Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3-4):335–363, 2000.
10. P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modeling of reactive systems using open nets. In K.G. Larsen and M. Nielsen, editors, *Proc. CONCUR 2001*, LNCS 2154, pages 502–518, 2001.
11. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comp. Prog.*, 19(2):87–152, 1992.
12. R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2):46–89, 2000.
13. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999.
14. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modeling of Workflows. In M.P. Papazoglou, editor, *Proc. 14th Int. Object-Oriented and Entity-Relationship Modelling Conference (OOER'95)*, LNCS 1021, pages 341–354. Springer, 1995.
15. S. Christensen and N. D. Hansen. Coloured Petri nets extended with channels for synchronous communication. Technical Report PB-390, Aarhus University, 1992.
16. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
17. 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. Available at `http://www.cs.utwente.nl/~tcm`.

18. W. Deiters and V. Gruhn. Process management in practice applying the FUNSOFT net approach to large-scale processes. *Autom. Softw. Eng.*, 5(1):7–25, 1998.
19. J. Desel and T. Erwin. Modeling, simulation and analysis of business processes. In Aalst et al. [1], pages 129–141.
20. J. Desel and G. Juhás. What is a Petri net? Informal answers for the informed reader. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets*, LNCS 2128, pages 1–27. Springer, 2001.
21. ebXML. `http://www.ebxml.org`.
22. 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*, LNCS 691, pages 1–16. Springer, 1993.
23. W. Emmerich and V. Gruhn. Software process modelling with FUNSOFT nets. Technical Report 47, Dept. of Computer Science, University of Dortmund, 1990.
24. H.-E. Eriksson and M. Penker. *Business Modeling With UML: Business Patterns at Work*. Wiley Computer Publishing, 2000.
25. R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. Int. Conf. on Software Eng. (ICSE)*, 2002. To appear.
26. R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001.
27. R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029, pages 76–90. Springer, 2001.
28. R. Eshuis and R. Wieringa. An execution algorithm for UML activity graphs. In M. Gogolla and C. Kobryn, editors, *Proc. ≪UML≫ 2001*, LNCS 2185, pages 47–61. Springer, 2001.
29. A. Foremniak and P.H. Starke. Analyzing and reducing simultaneous firing in signal-event nets. *Fundamenta Informaticae*, 43:81–104, 2000.
30. N. De Francesco, U. Montanari, and G. Ristori. Modelling concurrent accesses to shared data via Petri nets. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94*, pages 489–508. North-Holland, 1994.
31. H. J. Genrich. Predicate/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, LNCS 254, pages 207–247. Springer, 1987.
32. H. J. Genrich and P. S. Thiagarajan. A theory of bipolar synchronization schemes. *Theoretical Computer Science*, 30(3):241–318, 1984.
33. A. Geppert, D. Tombros, and K.R. Dittrich. Defining the semantics of reactive components in event-driven workflow execution with event histories. *Information Systems*, 23(3-4):235–252, 1998.
34. P. Grefen and R. Remmerts de Vries. A reference architecture for workflow management systems. *Journal of Data & Knowledge Engineering*, 27(1):31–57, 1998.
35. C. Hagen and G. Alonso. Beyond the black box: Event-based inter-process communication in process support systems. In *Proc. 19th Int. Conf. on Distributed Computing Systems (ICDCS '99)*, pages 450–457. IEEE, 1999.
36. H.-M. Hanisch and A. Lüder. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000.
37. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
38. D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985.
39. IBM. MQ Series Workflow (Websphere). `http://www.ibm.com`.

40. K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, 1992.
41. S. Joosten. Trigger Modelling for Workflow Analysis. In G. Chroust and A. Benczur, editors, *Proceedings CON'94*, pages 236–247, Vienna, October 1994.
42. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. FMOODS'99, IFIP TC6/WG6.1*, pages 331–347. Kluwer, 1999.
43. N.L. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.
44. Software Ley. Cosa. `http://www.cosa.de`.
45. F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques.* Prentice Hall, 2000.
46. M. Löwe, D. Wikarski, and Y. Han. Higher-order object nets and their application to workflow modeling. Technical Report 95-34, Informatik, Technical University Berlin, 1995.
47. U. Montanari and F. Rossi. Contextual nets. *Acta Inf.*, 32(6):545–596, 1995.
48. T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
49. A. Oberweis, R. Schätzle, W. Stucky, W. Weitz, and G. Zimmermann. INCOME/WF: A Petri net based approach to workflow management. In H. Krallmann, editor, *Wirtschaftsinformatik '97*, pages 557–580. Springer, 1997.
50. OMG. Workflow management facility specification. OMG Document Number formal/00-05-02, 2000. Available at `http://www.omg.org`.
51. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, 1981.
52. A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, LNCS 526, pages 244–265. Springer, 1991.
53. W. Reisig. *Petri Nets: An Introduction.* Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.
54. W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I: Advances in Petri nets*, LNCS 1491. Springer, 1998.
55. K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):218–230, 2001.
56. UML Revision Taskforce. *OMG UML Specification v. 1.4.* Object Management Group, 2001. Available at `http://www.omg.org`.
57. J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann Publishers, 1996.
58. R.J. Wieringa. *Design Methods for Software Systems: Yourdon, Statemate and the UML.* Morgan Kaufmann, 2002. To be published.
59. D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F.N. Afrati and P. Kolaitis, editors, *6th International Conference on Database Theory (ICDT)*, LNCS 1186. Springer, 1997.
60. Workflow Management Coalition. The workflow reference model (WFMC-TC-1003), 1995. Available at `www.wfmc.org`.
61. Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. Available at `www.wfmc.org`.