Open access • Book Chapter • DOI:10.1007/3-540-44919-1_20

Reactive Petri nets for workflow modeling — Source link

Rik Eshuis, Juliane Dehnert

Institutions: Technical University of Berlin

Published on: 23 Jun 2003 - Applications and Theory of Petri Nets

Topics: Workflow engine, Petri net, Operational semantics, Denotational semantics and XPDL

Reactive Petri Nets for Workflow Modeling

Rik Eshuis[1]* and Juliane Dehnert[2]

[1] LIASIT | CRP Henri Tudor
6 rue Coudenhove - Kalergi, L-1359 Luxembourg, Luxembourg
rik.eshuis@tudor.lu
[2] Institute for Computation and Information Structures (CIS),
Technical University Berlin,
Sekr.EN7, Einsteinufer 17, D-10587 Berlin, Germany
dehnert@cs.tu-berlin.de

Abstract. Petri nets are widely used for modeling and analyzing workflows. Using the token-game semantics, Petri net-based workflow models can be analyzed before the model is actually used at run time. At run time, a workflow model prescribes behavior of a reactive system (the workflow engine). But the token-game semantics models behavior of closed, active systems. Thus, the token-game semantics behavior of a Petri net-based workflow model will differ considerably from its behavior at run time. In this paper we define a reactive semantics for Petri nets. This semantics can model behavior of a reactive system and its environment. We compare this semantics with the token-game semantics and prove that under some conditions the reactive semantics and the token-game semantics induce similar behavior. Next, we apply the reactive semantics to workflow modeling and show how a workflow net can be transformed into a reactive workflow net. We prove that under some conditions the soundness property of a workflow net is preserved when the workflow net is transformed into a reactive workflow net. This result shows that to analyze soundness, the token-game semantics can safely be used, even though that semantics is not reactive.

1 Introduction

Petri nets are a popular technique to formally model workflows [1, 9, 11, 18]. They offer a formal counterpart for the bubbles and arrows that people draw when modeling workflows. Their formal token-game semantics enables analysis of Petri net based workflow models. Under the token-game semantics, the workflow model describes what behaviors are allowed. By computing behavior of a workflow model using the token-game semantics, errors in a workflow model can be spotted before the workflow model is actually put to use (cf. [1]).

A workflow model is put to use by feeding it to a workflow management system (WFMS). Heart of a WFMS is the workflow engine (WF engine), that does the actual management. WF engines are reactive systems. A reactive system

* Part of this work was done while the author was working at the University of Twente.

runs in parallel with its environment and tries to enforce certain desirable effects in the environment [17]. It does so by reacting to changes, called events, in its environment. The response of the reactive system depends upon its current state. Giving a response may change the state of the reactive system.

The WF engine sees a workflow model as a prescription of what it has to do. The behavior of reactive systems is usually modeled using event-condition-action (ECA) rules [19], also known as production rules. The meaning of an ECA rule is that if the event in the environment occurs, and the condition is true, the reactive system does the action. Part of the condition can be a test of the state of the system. Part of the action can be a change of state. ECA rules can be easily incorporated in Petri nets by associating with every transition in the Petri net an ECA rule, that tells how the transition changes the state of the reactive system, in this case the state of the WF engine. This insight is already present in the pioneering work on workflow modeling done in the seventies [21]. Thus, in this case a Petri net workflow model models the behavior of the WF engine.

Unfortunately, the token-game semantics of Petri nets does not model behavior of reactive systems, and therefore does not model behavior of a WF engine [12, 13]. The non-reactivity of the token-game semantics can be seen immediately from the definition of the firing rule. A transition in a Petri net is enabled once its input places are filled. The environment of the Petri net does not influence the firing of transitions. In contrast, in a reactive system a transition which is relevant, needs some additional input event to become enabled. So, the token-game semantics models closed systems, whereas a reactive system is open, otherwise it cannot interact with its environment.

Furthermore, in a reactive system an enabled transition *must* fire immediately, otherwise the system would fail to respond to a certain event. In the token-game semantics, an enabled transition *may* fire, but does not have to. In the worst case firing is postponed forever, or some transition that becomes enabled later fires before the enabled transition. Clearly, this contradicts reactivity.

To illustrate this point on a real-life example, consider the Petri net in Fig. 3. For an explanation of this example, we refer to Sect. 4. Now, suppose task check credit has just finished and that the new marking is [p1,p5]. Since task check credit has finished, presumably the WF engine now has to decide whether or not the credit was ok. (But the actual outcome of the decision depends upon the environment of the WF engine, which is modeled by the nondeterministic choice in place p5; see Sect. 4.) Then, under the token-game semantics, it is possible to fire transition check_order before firing transition ok or transition not ok. Suppose that the order is checked only after several days; then this firing sequence implies that the WF engine also takes days before it actually makes a decision. Clearly, this is inappropriate behavior: the decision should be made immediately when check credit has finished.

Since the token-game semantics is not reactive, a Petri net does not model the behavior of the WF engine. Thus, it is unclear how the behavior of the Petri net under the token-game semantics relates to the behavior of the WF engine when the workflow model is actually put to use. Consequently, it is also

unclear whether the outcome of analysis of a workflow model using the token-game semantics carries over to the reactive setting. In other words, are analysis results in which the token-game semantics has been used still valid in a reactive setting?

The purpose of this paper is to define a reactive semantics for Petri nets and to relate this new semantics to the standard Petri net token-game semantics. In particular, we will study in what respect and under what conditions these two semantics induce similar behavior. As an application of this result, we will show that the soundness property [1] is preserved when transforming a workflow net into a reactive workflow net, i.e., a workflow net with a reactive semantics. Thus, we give a justification why the token-game semantics can safely be used to analyse workflow models for absence of deadlocks in a reactive setting, even though the token-game semantics is not reactive.

There are some commercially Petri net based workflow management systems available. We do not claim that our reactive semantics precisely describes the behavior of a WF engine in such a WFMS, but we do think that our semantics is closer to the behavior of such a WF engine than the token-game semantics.

The remainder of this paper is structured as follows. In Sect. 2 we will recapitulate some standard terminology and notions from Petri net theory. In Sect. 3 we define a reactive semantics for Petri nets and relate this semantics to the standard token-game semantics. In particular, we prove that under some conditions both semantics induce similar behavior. In Sect. 4 we recall the definition of a workflow net [1] and the soundness property. We present different interpretations for transitions in a workflow net. Using these different interpretations, in Sect. 5 we show how a workflow net can be transformed in a reactive workflow net. In Sect. 6 we prove that the soundness property is preserved when a workflow net is mapped into a reactive workflow net. In the proof we build upon the results obtained in Sect. 3. Related work is discussed in Sect. 7. We end with conclusions and further work.

2 Preliminaries

We recall the definition of a Petri net (P/T net).

A Petri net is a triple (P, T, F), where

- P is a finite set of places,
- T is a finite set of transitions, $(P \cap T = \emptyset)$
- $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of arcs, the flow relation.

A transition t has input and output places. A place p is input (output) for transition t if there is a directed arc from p to t (from t to p). The input places of a transition t are denoted $\bullet t$. The output places of t are denoted $t\bullet$. A place can contain zero or more *tokens*. A token is represented by a black dot. The global *state* of a Petri net, also called a *marking*, is the distribution of tokens over places. Formally, a state or marking M is a function $M : P \to \mathbb{N}$ that assigns to every place p the number of tokens $M(p)$ that reside in p.

We now introduce some terminology.

– A transition t is *enabled* in marking M, written $M \xrightarrow{t}$, iff every input place of t contains at least one token.
– If a transition t is enabled in marking M, it *may fire*: from every input place one token is removed and to every output place one token is added. We write $M \xrightarrow{t} M'$ to denote that firing enabled transition t in marking M results in marking M'.

We write $M \rightarrow M'$ to indicate that by firing some transition t in M marking M' can be reached. We write $M \xrightarrow{\sigma} M'$ to denote that by firing sequence $\sigma = t_1 t_2 \dots t_n$ from M marking M' can be reached, so
$M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} M_2 \dots M_{n-1} \xrightarrow{t_{n-1}} M_n$, where $M_0 = M$ and $M_n = M'$. We write $M \xrightarrow{*} M'$ to denote that there is a sequence σ such that $M \xrightarrow{\sigma} M'$.

3 Reactive Nets

In this section we will adapt the definition of a Petri net to make it reactive. We call the new Petri net variant that we thus obtain a reactive net.

As explained in the introduction, the token-game semantics models closed systems, whereas a reactive system is open, otherwise it wouldn't be able to interact with its environment. This limitation of Petri nets can be circumvented by modeling the environment in the Petri net as well. We therefore distinguish external transitions of the environment from internal transitions of the reactive system. Thus, instead of a set T of transitions, we now have a set $T_{internal}$ of internal transitions and a set $T_{external}$ of external transitions.

We also explained in the introduction that the may firing rule used in Petri nets does not model reactivity. In a reactive system an enabled transition *must* fire immediately, otherwise the system would fail to respond to a certain event. In the token-game semantics, an enabled transition t *may* fire, but does not have to. In the worst case firing t is postponed forever, or some conflicting transition that becomes enabled later fires before t and disables t.

The most straightforward way to make the token-game semantics reactive is to change the firing rule from may firing into must firing, i.e., as many enabled transitions as possible should fire. This, however, is undesirable: since Petri nets model closed systems, the environment of the system is also included in the Petri net. The environment is active rather than reactive. For the environment, the may firing rule is more appropriate. Therefore, for internal transitions, done by the reactive system itself, we use a must firing rule, and for external transitions, done by the environment, we use a may firing rule.

However, if in a certain marking both an internal transition and an external transition are enabled, a conflict can arise. To avoid such a conflict, we require all internal transitions to fire with higher priority than external transitions. So, if both the environment and the reactive system can do a transition, the reactive system will fire first. This corresponds to the perfect synchrony hypothesis [5], an assumption frequently made in the design of reactive systems: The reactive system is faster than the environment it controls. Note that the perfect synchrony hypothesis is an assumption, not a guarantee.

We can informally describe the behavior of reactive Petri nets in the following way, borrowing some terminology from STATEMATE [16]. A state (marking) is *stable* if no internal transition is enabled, it is unstable otherwise. A stable state can become unstable if some external transition fires. In an unstable state, the reactive system must fire some enabled internal transitions. By firing these transitions, a new state is reached. If the new state is stable, the system has finished its reaction. Otherwise, the system again reacts by taking a transition and entering another new state. This sequence of taking a transition and entering a new state is repeated until finally a stable state is reached. It is possible that the system never reaches such a stable state: in that case the system diverges.

Definition. We now describe reactive nets and their behavior more formally. A *reactive net RN* is a tuple $(P, T_{internal}, T_{external}, F)$. Sets $T_{external}$ and $T_{internal}$ are transitions. A reactive semantics for a net $(P, T_{internal}, T_{external}, F)$ is defined as follows.

- An internal transition t is enabled in marking M iff all of t's input places are filled with a token, i.e. M is unstable.
- An external transition t is enabled in marking M iff all of t's input places are filled with a token, and there is no enabled internal transition in M, i.e. M is stable.

The firing of an enabled transition is as before: from every input place one token is removed and to every output place one token is added.

The must firing is encoded in the priority rule: enabled internal transitions have priority over external transitions. That unstable markings are instantaneous is an interpretation we attach to them. To model this explicitly in the semantics, we would have to switch to timed Petri nets.

Token-game semantics for reactive nets. It also possible to use a token-game semantics for a reactive net, by first transforming the reactive net into a Petri net using function *toPetri*, which takes the union of sets $T_{internal}$ and $T_{external}$. Function *toPetri* is defined as follows:

$$toPetri((P, T_{internal}, T_{external}, F)) = (P, T_{internal} \cup T_{external}, F)$$

To show the relation between the transition relation of RN under the token-game semantics, $\rightarrow^{tg}$, and the transition relation of RN under the reactive semantics, $\rightarrow^{r}$, we now define $\rightarrow^{r}$ in terms of $\rightarrow^{tg}$:

$$\begin{aligned} M \rightarrow^{r} M' \Leftrightarrow\ & \exists t_i \in T_{internal} : M \xrightarrow{t_i}{}^{tg} M' \\ & \vee\ (\ (\exists t_e \in T_{external} : M \xrightarrow{t_e}{}^{tg} M') \wedge (\nexists t_i \in T_{internal} : M \xrightarrow{t_i}{}^{tg})\ ) \end{aligned}$$

Relation between reactive and token-game semantics. Indirectly, we have provided two different semantics for reactive nets: a reactive one and the traditional token-game semantics. In what respect do these semantics induce similar behavior?

Before we answer this question, let us look at the behavior of the reactive system in the reactive setting. In the reactive setting, the system moves typically from a stable state to another stable state. It is also possible that the system diverges: then there is a loop of internal transitions in the reactive net.

Of course, this reactive behavior can be simulated with the token-game semantics, since every transition enabled in marking M under the reactive semantics will also be enabled in M under the token-game semantics. The next theorem and corollary now follow immediately.

Theorem 1 *Given a reactive net RN that is in some state M. If under the reactive semantics t can fire in M and M' is reached, $M \xrightarrow{t}{}^{r} M'$, then t can also fire under the token-game semantics and M' is also reached, so $M \xrightarrow{t}{}^{tg} M'$.*

Corollary 2 *Given a reactive net RN. If M is a reachable state of RN under the reactive semantics, then M is a reachable state of RN under the token-game semantics.*

Clearly, the token-game semantics can be used to simulate any reaction, starting in some stable state, in the reactive semantics. The reverse, however, does not hold: not every behavior under the token-game semantics can be simulated using the reactive semantics. The reason for this is that in the reactive semantics external transitions are only enabled once all enabled internal transitions have fired. So, not every marking reachable under the token-game semantics will be reachable under the reactive semantics.

However, sometimes the *outcome* of the reaction of the system in the two semantics, i.e., the stable marking that is eventually reached, can be the same in both semantics. That is, under some constraints, if under the token-game semantics a particular stable marking is reached (where stable under the token-game semantics means that no internal transition is enabled), then this same marking can be reached under the reactive semantics. The constraints needed to enforces this, C1 and C2, are listed in Table 1.

Constraint C1 is necessary because under the reactive semantics internal transitions have priority over external transitions, whereas under the token-game semantics this is not the case. To see why constraint C2 is needed, consider the example reactive Petri net in Fig. 1. This reactive net does not satisfy C2, since t6 conflicts with both t2 and t4. Under the token-game semantics, in stable marking [p2, p3], there is a sequence t3, t5, t6 to stable marking [o]. All the intermediary markings in the sequence are unstable. Yet, it is impossible in the

Table 1. Constraints on reactive nets

C1	An external transition t_e does not conflict with an internal transition t_i: $\bullet t_i \cap \bullet t_e \neq \emptyset$.
C2	For two internal transitions t and t', if $\bullet t \cap \bullet t' \neq \emptyset$, then either $\bullet t = \bullet t'$ (t and t' are free choice), or there is no reachable marking M, under the token-game semantics, such that $M \xrightarrow{t}{}^{tg}$ and $M \xrightarrow{t'}{}^{tg}$.

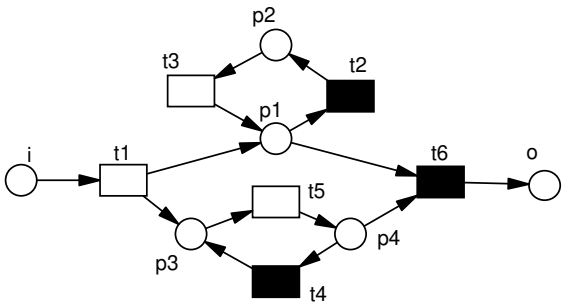

Fig. 1. Example reactive net to motivate constraint C2. Black transitions are internal, white transitions are external

reactive semantics to do such a sequence. Transition t6 is never taken; instead, t2 or t4 is taken. So marking [o] is unreachable under the reactive semantics.

Constraint C2 has been deliberately formulated on both the syntax and semantics of Petri nets, rather than only on the syntax. A syntactic constraint, e.g. "internal transitions are free choice", would have ruled out certain reactive Petri nets that constraint C2 allows. Figure 3 shows a Petri net, in which the two internal transitions cancel and pick satisfy constraint C2, but are not free choice.

We now proceed to prove one of the main theorems of this paper. The theorem states that under constraints C1 and C2, the token-game semantics and the reactive semantics have similar behavior. In the theorem, we use the terms 'stable' and 'unstable' states, introduced before.

Theorem 3 *Given a reactive net RN that satisfies constraints C1 and C2.*

Suppose, under the token-game semantics, there is a sequence σ of transitions from one stable state to another, such that all intermediary states are unstable,

$$M_0 \xrightarrow{t_0}{}^{tg} M_1 \xrightarrow{t_1}{}^{tg} M_2 \xrightarrow{t_2}{}^{tg} M_3 \xrightarrow{t_3}{}^{tg} \ldots \xrightarrow{t_{n-1}}{}^{tg} M_n$$

where M_0 and M_n are stable, and M_i, for $0 < i < n$, is unstable.

Then a permutation of σ (possibly σ itself) can be taken under the reactive semantics, and stable state M_n is reached.

Proof. If sequence σ is possible under the reactive semantics, we are done. So assume that σ is not possible under the reactive semantics. Then in some unstable state M_i an external transition $t_{external}$ is taken (so $t_i = t_{external}$). Under the reactive semantics, $t_{external}$ is disabled in M_i. Since M_i is unstable, there must be some enabled internal transitions in M_i. By C2 and since M_n is stable, one of these internal transitions, say t, is taken somewhere later in the sequence in some state M_j, where $j > i$. By C1, t and $t_{external}$ do not disable each other: the tokens in t's input places are not removed if $t_{external}$ is taken and vice versa. We modify σ by removing t_j from σ and inserting t just before $t_{external}$. Denote this

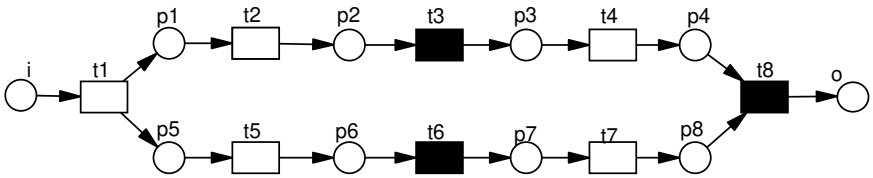

Fig. 2. Example reactive net to illustrate Theorem 3. Black transitions are internal, white transitions are external

modified sequence by σ'. Clearly, σ' can be taken under the token-game semantics; state M_n is then reached. Of course, sequence σ' may not be possible under the reactive semantics, because in some unstable state an external transition is taken. Then again the procedure sketched above has to be applied.

Thus, by repeatedly applying the procedure sketched above, finally a sequence σ_{final} is obtained. (The procedure terminates because the sequence is finite, and internal transitions are given a place earlier in the sequence.) In σ_{final}, in every unstable state an internal transition is taken. So σ_{final} can be taken in the reactive semantics. □

Example. Consider the example reactive net in Fig. 2. It satisfies constraints C1 and C2. Under the token-game semantics, a possible sequence from stable state [p1,p5] to stable state [p3,p7] is t2,t5,t6,t3. All intermediate states in this sequence are unstable. This sequence cannot be taken under the reactive semantics, because in unstable state [p2,p5] external transition t5 is fired. By applying the procedure sketched in the proof, we obtain sequence t2,t3,t5,t6. This sequence can be taken under the reactive semantics. Note that this sequence has an intermediate stable state [p3,p5] not present in the original sequence.

4 Workflow Nets

In this section we recall the definition of Workflow nets [1] and give different interpretations for transitions in Workflow nets.

Definition. A *Workflow net* (WF net) is a Petri net with one input place i and one output place o such that:

- Place i does not have incoming arcs.
- Place o does not have outgoing arcs.
- Every node $n \in P \cup T$ is on a path from i to o.

WF nets use the standard Petri net token-game semantics.

Figure 3 shows a WF net for handling an incoming order for a mobile telephone. It is a reduced version of a real-life business process of a telephone company. The process involves two departments: the accountancy department, which handles the payment, and the sales department, which handles the distribution.

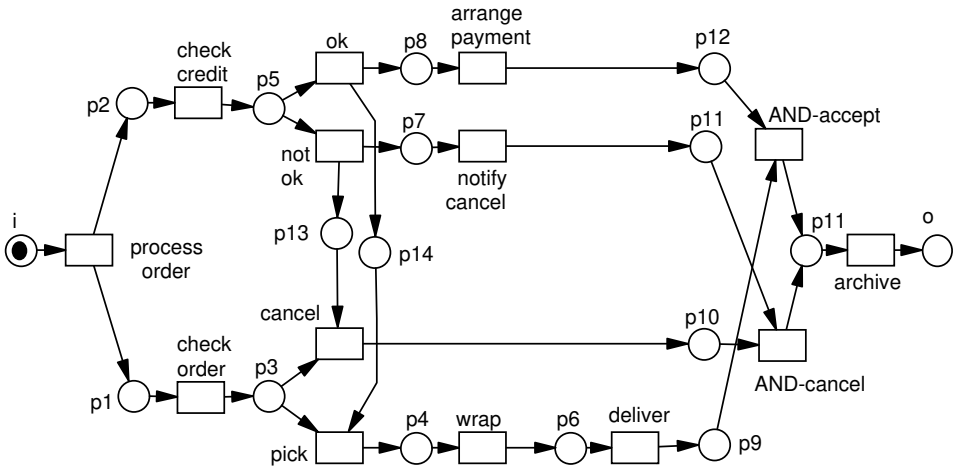

Fig. 3. WF net modeling the process "Handling of incoming order"

The process starts with an incoming order that is processed to the different departments. This is modeled by transition process_order which splits the execution into two parallel threads. The bottom part models the tasks on the sales side. Here the order is handled (executing tasks check_order, pick, wrap, and deliver). The top part models the tasks on the accounting side. Here the customer standing is checked first (check_credit). The result of this task is either ok or not_ok. In case the result is positive the payment is arranged (arrange_payment), in the latter case the order is refused (notify_cancel).

The cooperation of the two departments follows a pessimistic strategy. The sales department always waits for the outcome of the credit check performed by the accountancy. Depending on the outcome it either picks, wraps, and delivers the item or cancels it further processing.

Transitions in WF nets. In a WF net states are modeled via places, whereas transitions model active behavior. Transitions are used for different purpose. In the most common case they are used to model *activities* (or *tasks*). Examples of tasks in the process of Fig. 3 are check_order, pick, wrap, deliver, cancel, check_credit, arrange_payment, notify_cancel and archive.

Sometimes, transitions represent the making of *decisions*. Examples are the two transitions ok, not_ok representing the outcome of the task check_credit. Note that the outcome of the decision is determined by the environment of the WF engine; we come back to this issue below.

Transitions may also be employed to depict the occurrence of external *events*. Figure 4 gives an example. The WF net models part of the library process for returning books. Upon borrowing books, the system waits for an external event: This may either be the reader bringing the books back, or the reader asking for extension, or a timeout. Depending on the particular event occurring, a following task is executed.

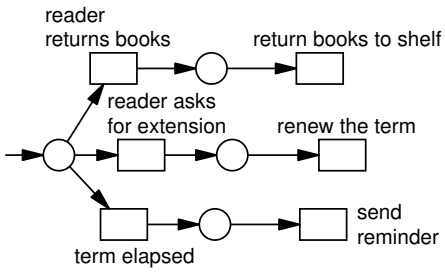

Fig. 4. Part of the library process "Return Books"

Finally, transitions can be there for the sole purpose of *routing* a case. This usually occurs when a case needs to be split into parallel parts (fork) or parallel parts of a case need to be merged (join). Examples for this case are process_order, AND_cancel, and AND_accept from Fig. 3.

Function *type* assigns to each transition the purpose of the transition.

$$type : T \rightarrow \{task, event, decision, routing\}$$

Van der Aalst [1] distinguishes task and decision transitions, but does not treat them differently. In WF nets, all transitions use the same firing rule. So there is no distinction between the different types of transitions. In Sect. 5, we will attach different semantics to these different types of transitions.

There exist some interesting dependencies between transitions of different type. Transition of type *decision* always follow a *task* transition. This task transition models the processing of some kind of test. For the evaluation it refers to external data. Transitions of type *decision* always occur within a choice. They are in conflict. Furthermore, it can be assumed that the corresponding choice is free choice. This means that only the evaluation of the external information (and nothing else) decides about the outcome of the choice.

If transitions of type *event* occur within a choice, we assume that choice to be free choice as well. This is a reasonable assumption as the occuring events should preclude each other. This means that the particular external event that occurs (and nothing else) decides the outcome of the choice. For instance, in the library example from Fig. 4 the timeout event occurs only if the reader neither returns the book (in person) nor asks for extension (e.g. via email or telephone). Depending on the kind of external event the books are either returned to the shelf, the term is renewed, or a reminder is send.

The *outcome* of the above mentioned choices depends either on the evaluation of external data or the event occurring. In [7] these choices have been clustered using the notation of a *non-controllable choice*. This notation suggests that the *outcome* of these choice depends on the environment. Choices whose outcome do not depend on the environment are called *controllable*; they can be controlled by the WF engine.

Table 2. Choice classification

	initiative	**outcome**
choice of *event* transitions (free choice)	environment	environment
choice of *decision* transitions (free choice)	WF engine	environment
choice of *task* and/or *routing transitions*	WF engine	WF engine

An orthogonal criterion to distinguish choices, is the moment of choice, i.e., the moment one of the alternative transitions is executed. This distinction was made by Van der Aalst and his coworkers [1, 3]. They distinguish between implicit and explicit choices. An implicit choice (also deferred choice) is made the moment an external event occurs, hence it corresponds to a choice that consists of *event* transitions. An explicit choice is made the moment the previous task is completed. In our framework, explicit choices correspond to choices consisting of *decision* transitions.

In our framework, the moment of choice is determined by the one having the *initiative* to execute a transition. Events depict behavior of the environment, hence the *initiative* to execute an *event* transition is at the environment. In contrast, the initiative for transitions of type *decision* is at the side of the WF engine. The WF engine executes these transitions. In Sect. 5, we will map WF nets to reactive nets. *Event* transitions will be external, whereas *decision* transitions will be internal. Thus, in the reactive setting, an implicit choice behaves differently from an explicit choice.

Remaining choices are choices that consist of transitions of type task and of type routing. These are always controllable and explicit choices. These choices are furthermore not necessarily free choice. An example is the choice between the task transitions pick and cancel in Fig. 3.

Table 2 summarizes the possible influences of WF engine and environment on different choices.

Correctness criteria for WF nets. We only consider soundness as introduced by Van der Aalst [1]. Soundness requires that a WF net can *always* terminate with a single token in place o and that all the other places are then empty. In addition, it requires that there is no dead transition, i.e. each transition can be executed. We recall the definition of soundness as defined in [1]. To stress that soundness is defined on the token-game semantics, we label the transition relation with tg.

Definition 4. *(Soundness) A WF net is sound iff:*
(i) For every marking M reachable from marking i, there exists a firing sequence leading from marking M to marking o.

$$\forall_M([i] \xrightarrow{*}{}^{tg} M) \Rightarrow (M \xrightarrow{*}{}^{tg} [o]).$$

(ii) Marking $[o]$ is the only marking reachable from marking $[i]$ with at least one token in place o (proper termination).

$$\forall_M([i] \xrightarrow{*}{}^{tg} M \wedge M \geq [o]) \Rightarrow (M = [o])$$

(iii) The WF net does not have dead transitions.

$$\forall_{t \in T} \exists_{M,M'} ([i] \xrightarrow{*}{}^{tg} M \xrightarrow{t}{}^{tg} M')$$

If a WF net is going to be used as input for a WF engine, soundness is essential. In a sound WF net every firing sequences terminates properly. Deadlocking executions, as well as executions where spare tokens remain in the net, are impossible. If the process description is used as base for operation at run time, soundness is a necessary requirement in order to guarantee a reliable execution.

However, the soundness criterion is defined using the token-game semantics. As we saw in the introduction, that semantics is not reactive. Therefore, in the next section we show how to transform a WF net into a reactive WF net. In Sect. 6 we show that the transformation preserves soundness. For reactive WF nets, soundness is defined by replacing in the definition above $\rightarrow^{tg}$ with $\rightarrow^{r}$.

5 From Workflow Nets to Reactive Workflow Nets

Workflow management is a current issue in many business (re-)engineering projects. It comprises support for the modeling, the analysis and the run-time execution of business processes. Many approaches aiming at providing support for workflow management are based on the use of Petri nets (e.g. [1, 8, 9]) or were mapped onto Petri nets (e.g. [2, 8]). Even though these approaches cover the modeling and the analysis of business processes, they provide only limited support for the execution at run time. Reasons for that gap have been discussed in the introduction and concern the mismatch between the reactive behavior of the workflow (WF) engine and the active behavior of Petri nets (and thus WF nets) using the token-game semantics.

In this section we discuss how a WF net can be transformed into a reactive WF net. A reactive WF net is a Petri net with a reactive semantics. It can serve as input for a WF engine specifying what the WF engine should do.

The following three properties of WF nets makes that they are not entirely suitable as input for a WF engine.

1. Transitions fire instantaneously. This does not match with the requirement to model tasks as time consuming entities.
2. Usually, transitions in a WF net model tasks. The WF engine monitors tasks, but does not do them. Thus, it is hard to detect from the WF net the actual behavior of the WF engine.
3. Under the token-game semantics, a Petri net models an active system. But a WF engine is a reactive system. For such a system, the may-firing rule of the token-game semantics introduces unintended non-determinism, allowing either to execute an enabled task or to defer its execution.

We now show how we can overcome these three obstacles. We will change the perspective of a WF net from modeling a process to monitoring it. This way, we obtain a description of the desired behavior of a WF engine.

Task refinement. Recall from Sect. 4 that transitions in a WF net either model the occurrence of an event (type: *event*), the making of a decision (type: *decision*), the routing of tasks (type: *routing*) or the actual task that is executed by some external actor (type: *task*). The firing of transitions is considered to be instantaneous. This abstraction is adequate for transitions that model events, decisions or routing, but this generalization does not fit for tasks.

Changing the perspective from active task execution to only monitoring it, tasks performed by external actors should be modeled as time consuming. We therefore refine the modeling and depict a task as a sequence of transitions announce_task, begin_task, end_task, and record_task_completion. Figure 5 illustrates the described task refinement. The transition announce_task models the placing of the task to a possible actor. This may either mean that it is "pushed" into someones in-basket or that the task is put to a common list, from where it can be "pulled" by any actor. The precise implementation depends on the mode of the WFMS.

The actual processing of the task starts with transition begin_task and ends with end_task. This way the instantaneous firing of transitions can be retained now matching an acceptable abstraction. Note that the duration implicitly assigned to the execution of a task in a WF net is now assigned to a place in the refined WF net.

Division of powers between the WF engine and the environment. Changing the perspective of a WF net towards monitoring we have to distinguish precisely between the behavior of the WF engine and the environment. In WF nets, such a dinstinction is not made.

From a monitoring perspective, a transition is either executed by the WF engine, or executed by the environment of the WF engine. Taking the perspective of the WF engine, we call transitions executed by the WF engine *internal*, whereas transitions by the environment are *external*. We will therefore split the set of transitions T of a WF net $PN = (P, T, F)$ into disjoint sets of internal and external transitions: $T = T_{internal} \cup T_{external}$. Internal transitions are denoted by black whereas external transitions are represented by white boxes.

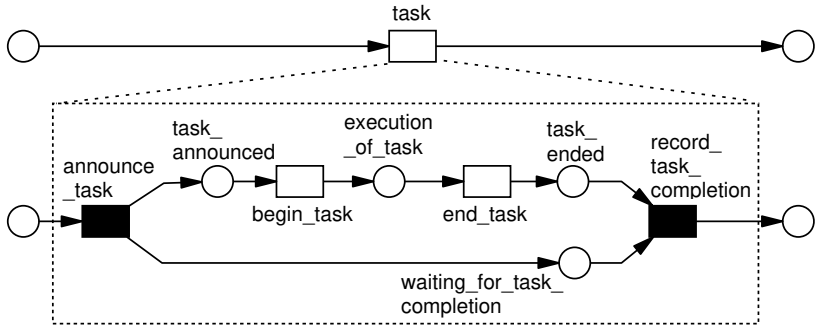

Fig. 5. Task refinement

Table 3. Mapping WF net to reactive WF net

transition type	internal	external
announce_task	x	
begin_task		x
end_task		x
record_task_completion	x	
event		x
decision	x	
routing	x	

Reviewing the four different transition types: task, event, decision, routing, we can classify transitions as internal or external as follows (see Table 3):

Tasks The announce_task and record_task_completion transitions are internal and the begin_task and end_task transition are external. This denotes that the WF engine initiates the task but that an actor outside the WF engine does the actual task. The WF engine waits for completion of the task.

Events These transitions are external. This is natural as such transitions model the occurrence of events coming from the environment.

Decisions These transitions are internal. They are done by the WF engine, even though the outcome of the decision presumably depends upon the task that has been executed immediately before (cf. Sect. 4, in particular Table 2).

Routing These transitions are internal, as routing is done by the WF engine.

Reconsidering Table 2, we can see that the party (WF engine or environment) having the initiative in taking a transition, also executes the transition.

Reactive semantics: Changing the firing rule. The last obstacle concerns the may firing rule of the token-game semantics. This rule states that if a transition t is enabled it may fire but does not have to. In the worst case, some conflicting transition that become enabled later fires before t, disabling t. This firing rule is not adequate to model behavior of the WF engine, which is modeled through internal transitions.

We therefore transform a refined WF net (P, T, F) into a reactive WF net $(P, T_{internal}, T_{external}, F)$, using the previously introduced distinction between internal and external transitions (Table 3). This reactive WF net can be mapped to the original refined WF net using function *toPetri*. Thus, we replace the token-game semantic of the refined WF net by a reactive semantics, i.e. replace relation $\rightarrow^{tg}$ by $\rightarrow^{r}$ (see Sect. 3).

To illustrate this mapping, Fig. 6 shows the reactive WF net corresponding to the example WF net of Fig. 3. Due to space limitations, we do not show how task transitions are refined, but just depict them with a shortcut: a transition subdivided into three sections: start and end black, middle white.

Note. We stated that reactive WF nets can be used by a WF engine to control and monitor processes. But not every transition of the reactive WF net

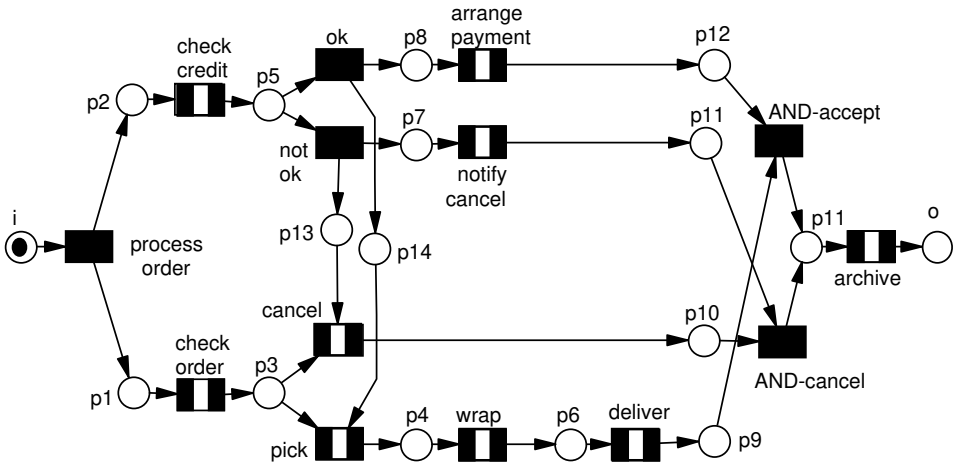

Fig. 6. A reactive WF net. Black transitions are internal, white transitions are external. Task transitions are decomposed as in Fig. 5

is executed by the WF engine; only internal transitions are. Thus, for a WF engine, a reactive WF net still contains too much information. By removing external transitions and external places (i.e. places filled by external transitions) from the WF net, a prescription for the WF engine is obtained. Note that by filling some places (e.g. place task_ended) in Fig. 5) with a token, the environment can trigger the WF engine to start doing something (e.g. do transition record_task_completion). Finally, we observe that reactive WF nets presupposes that the WF engine is faster than its environment, i.e., the WF engine must satisfy the perfect synchrony hypothesis (see Sect. 3).

6 Soundness of Reactive Workflow Nets

In the previous section we have defined a mapping from WF nets to reactive WF nets in two steps. First, we have refined task transitions to multiple transitions. Second, we have mapped a refined WF net to a reactive WF net. In this section, we show that the soundness property of a WF net is preserved when the WF net is mapped into a reactive WF net, provided the WF net meets the constraints defined in Table 1. (Although the constraints in Table 1 are not defined on WF nets, they can be lifted to WF nets by using the mapping defined in Table 3, provided function *type* is defined. Note that the task refinement (Fig. 5) satisfies the constraints C1 and C2.)

Given a WF net PN, we denote its refined variant by PN_{ref}. We denote the reactive variant of PN_{ref} by $PN_{reactive}$. For PN_{ref} we use the token-game semantics; for $PN_{reactive}$ the reactive semantics.

Theorem 5 *Sound* $PN \Leftrightarrow$ *sound* PN_{ref}.

Proof. Straightforward. □

The following theorem shows that if a reactive WF net is sound, the WF net is sound as well. Note that the soundness property (see Sect. 4) is defined on the token-game semantics, not on the reactive semantics. To obtain the soundness property for reactive WF nets, replace in the definition $\rightarrow^{tg}$ by $\rightarrow^{r}$.

Theorem 6 *$PN_{reactive}$ is sound $\Rightarrow$ PN_{ref} is sound.*

Proof. Follows immediately from Theorem 1 and Corollary 2. □

We now prove one of the main theorems of this paper. In the proof, we make use of the terminology of stable and unstable states, introduced in Sect. 3. State $[o]$ is stable by definition. We also assume state $[i]$ is stable. (It is possible to relax this constraint, but it will make the proofs more difficult.) To prove this theorem, we build on Theorem 3, so we need the constraints defined in Table 1.

Theorem 7 *Assume PN_{ref} (and thus $PN_{reactive}$) satisfies constraints C1 and C2. Then PN_{ref} is sound $\Rightarrow$ $PN_{reactive}$ is sound.*

Proof. We consider the three cases of the definition of soundness.

(i) Let M be an arbitrary state in $PN_{reactive}$. By Corollary 2, M is also reachable in PN_{ref}. Since PN_{ref} is sound, there is at least one firing sequence from M to o under the token-game semantics. Denote this sequence by σ. There are two cases.
- If M is stable, then σ can be split in subsequences σ_0, σ_1, ..., σ_n, such that $M \xrightarrow{\sigma_0}{}^{tg} M_1 \xrightarrow{\sigma_1}{}^{tg} M_2 \xrightarrow{\sigma_2}{}^{tg} \ldots \xrightarrow{\sigma_n}{}^{tg} [o]$, where M_1, M_2, ..., $M_n = [o]$ are all stable, and all other states visited in the sequence σ are unstable. By applying Theorem 3 on the sequences $\sigma_0, \sigma_1, \ldots, \sigma_n$, we have that there are permutations of these sequences $\sigma'_0, \ldots, \sigma'_n$ such that $M \xrightarrow{\sigma'_0}{}^{r} M_1 \xrightarrow{\sigma'_1}{}^{r} M_2 \xrightarrow{\sigma'_2}{}^{r} \ldots \xrightarrow{\sigma'_n}{}^{r} [o]$.
- If M is unstable, then there is at least a sequence of transitions from some stable state M_0 to stable state M_n that leads through M. (There is at least one sequence from $[i]$ to $[o]$.) For $M_n \xrightarrow{*}{}^{r} [o]$, we can argue as in the previous case. The remainder then follows easily.

(ii) Every reachable state in $PN_{reactive}$ is also a reachable state in PN_{ref} (Corollary 2). So if a state M with $M > [o]$ would be reachable in $PN_{reactive}$ it would also be reachable in PN_{ref}. But PN_{ref} is sound. So we have a contradiction.

(iii) For every transition t, there is a state M in PN_{ref}, such that $M \xrightarrow{t}$. In PN_{ref}, there is at least one sequence σ of transitions, one of which is t, from some stable state M_0 to a stable state M_n, which passes M. (There is at least one sequence from $[i]$ to $[o]$.) By Theorem 3, a permutation of σ can be taken in $PN_{reactive}$. So t can be taken in $PN_{reactive}$. Therefore, there are no dead transitions in $PN_{reactive}$.

□

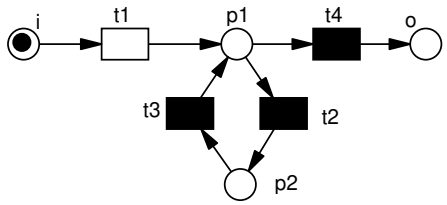

Fig. 7. Sound reactive WF net that can diverge. The net satisfies constraints C1 and C2. Black transitions are internal, white transitions are external

The desired result, that soundness is preserved when transforming a WF net into a reactive net, now follows immediately.

Corollary 8 *Assume PN_{ref} (and thus $PN_{reactive}$) satisfies constraints C1 and C2. Then PN is sound $\Leftrightarrow$ $PN_{reactive}$ is sound.*

Proof. By Theorem 5, we have: PN is sound $\Leftrightarrow$ PN_{ref} is sound. By Theorems 6 and 7, we have: PN_{ref} is sound $\Leftrightarrow$ $PN_{reactive}$ is sound. □

Finally, we note that soundness of a reactive net does not guarantee that the net is divergence free. A reactive net diverges if there is a loop of internal transitions. It is easy to prove that in a sound reactive net satisfying constraints C1 and C2, a loop of internal transitions can always be exited by taking some internal transition t that leaves the loop. However, soundness only states that t can be taken, but does not guarantee that t *will* be taken if the system is in a loop. For example, in Fig. 7, the system can do t1,t2,t3,t2,t3,..., staying forever in loop t2, t3 without ever taking t4. So, to guarantee absence of divergence, an additional constraint is needed. A sufficient, but not necessary, constraint is to require that the input places of internal free choice transitions are only filled by external transitions. That constraint would rule out the reactive WF net in Fig. 7.

7 Related Work

Like the present work, Wikarski [20] argues that the may firing rule is not suited to prescribe behavior of a system. He therefore proposes to use the may firing rule when the Petri net describes behavior, and the must firing rule when the net prescribes behavior. Thus, a net either uses may or must firing but not both. In contrast, reactive nets are both descriptive (external) and prescriptive (internal transitions); reactive nets use a mixture of may and must firing.

Next, we look at other work that uses Petri nets for modeling reactive systems. Then we look at some related work done on priority nets. We also discuss the reactive semantics one of the authors has defined for UML activity diagrams.

Petri nets for modeling reactive systems. In the past, several other extensions for Petri nets have been proposed, that, like the present work, have been motivated

by the desire to use Petri nets to control a process. We discuss two of those, Signal Event nets [15, 14] and Grafcet [6].

Signal event nets [15, 14] were introduced to model the combined behavior of some process and the controller that controls and monitors that process. Like in reactive nets, in Signal Event nets some transitions are by the environment (these are called spontaneous) whereas others are by the controller (these are called forced). Spontaneous transitions trigger forced transitions through transition synchronization. This way, it can be specified that a controller reacts to events in the environment. In reactive nets, an external transition triggers an internal transition t_i indirectly by filling all of t_i's input places.

Grafcet [6] is a standardized graphical language for specifying logic controllers. In Grafcet, Petri nets are extended with boolean variables, that are set by the environment of the system. These boolean variables represent the state of the environment. Every transition of a Grafcet model is a transition by the reactive system (i.e., the logic controller). Each transition has a corresponding event-condition-action (ECA) rule. If the event occurs and the condition is true and its input places are filled, the transition must fire immediately. Multiple transitions can fire simultaneously at the same time. In contrast, reactive nets use an interleaving firing rule.

There are some general differences between these approaches and reactive nets. First, reactive nets stay closer to the token-game semantics than these related approaches. Moreover, the definition of reactive nets is considerably simpler than the definition of signal-event nets and Grafcet models. Finally, none of these approaches attach a token-game semantics to a reactive model to relate it to the reactive semantics.

Priority nets. Reactive nets resemble Petri nets with priorities. The work on priority nets most close to ours is Bause [4]. Like the present work, Bause extends a net by defining a static priority ordering on its transitions. He shows that under a certain condition, properties like liveness of a net are preserved when the net is extended with static priorities. Bause's condition is similar to C1 and the constraint that internal transitions are free choice. However, our constraint C2 is more general, allowing internal transitions that are not free choice. For example, the Petri net in Fig. 3, does not satisfy Bause's condition, whereas it does satisfy our constraints C1 and C2. Motivated by the domain of stochastic Petri nets, Bause considers weighted Petri nets whose transitions are partitioned into multiple priority classes, whereas we, motivated by the domain of reactive systems, only consider unweighted Petri nets whose transitions are partitioned into two priority classes (internal and external transitions).

UML activity diagrams. Recently, one of the authors has defined two formal, reactive execution semantics for activity diagrams [12]. The design choices in both semantics are based on existing statechart semantics. The token-game semantics was not used, because that semantics is not reactive [12, 13]. Even though reactive nets are more reactive than Petri nets having a token-game semantics,

there still exist a lot of subtle differences between activity diagrams and reactive nets. For example, activity diagrams can refer to temporal events. These cannot be modeled in reactive nets; we would have to switch to timed Petri nets to model this. Moreover, activity diagrams can have data, whereas reactive nets can not. (Data can be modeled by switching to colored nets, but activity diagrams also differ from colored Petri nets [12, 13].) Consequently, in reactive WF nets, conflicting decision transitions (for example ok and not ok in Fig. 3) are enabled at the same time: the net behaves non-deterministically, whereas in activity diagrams such decisions are deterministic.

8 Conclusion and Further Work

We have defined a reactive variant of Petri nets, called reactive nets. Reactive nets assume that the systems they model are perfectly synchronous. Reactive nets have a reactive semantics, which differs from the token-game semantics, but they also can use the traditional token-game semantics. We have shown that under some conditions, the reactive and token-game semantics induce similar behavior, i.e., the same stable states are eventually reached.

Reactive nets are motivated by the domain of workflow management. We have shown how a WF net can be transformed into a reactive WF net, and that under some conditions the soundness property is preserved. Thus, we have offered a justification why soundness can be analyzed on WF nets using the token-game semantics, even though that semantics does not model behavior of reactive systems, whereas a WF engine is reactive. However, our work shows that in addition to soundness some extra constraints are needed. Moreover, soundness does not rule out divergence. An interesting topic of future work is to investigate whether the extra constraints are not too restrictive, i.e., whether they cover a large class of workflow models.

Another topic of future work is extending the semantics of reactive nets with simple real-time constructs, or addition of data. Next, it might be interesting to see whether the definition of reactive nets can be changed such that the perfect synchrony hypothesis is no longer needed.

Acknowledgements. The work of the first author has been partially supported by the National Research Fund (FNR) of Luxembourg and has been partially performed within the scope of the LIASIT (Luxembourg International Advanced Studies in Information Technologies) Institute.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst, A. Hirnschall, and E. Verbeek. An alternative way to analyze workflow graphs. In *Proc. 13th Int. Conference on Advanced Information Systems Engineering (CAiSE 2002)*, volume 2348 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

3. W.M.P. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuremann, editors, *Proc. 7th IFCIS Int. Conference on Cooperative Information System (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer Verlag, 2000.
4. F. Bause. On the analysis of Petri nets with static priorities. *Acta Informatica*, 33(7):669–685, 1996.
5. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. R. David. Grafcet: A powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, 3(3):253–267, 1995.
7. J. Dehnert. Non-controllable choice robustness: Expressing the controllability of workflow processes. In J. Esparza and C. Lakos, editors, *Proc. 23rd Int. Conference on Application and Theory of Petri Nets (ICATPN 2002)*, volume 2360 of *Lecture Notes in Computer Science*, pages 121–141. Springer Verlag, 2002.
8. J. Dehnert. Four steps towards sound business process models. In Ehrig et al. [10]. To appear.
9. J. Desel and T. Erwin. Modeling, simulation and analysis of business processes. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.
10. H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Petri Net Technology for Communication Based Systems*, Lecture Notes in Computer Science. Springer Verlag, 2003. To appear.
11. C.A. Ellis and G.J. Nutt. Modelling and enactment of workflow systems. In M. Ajmone Marsan, editor, *Proc. 14th Int. Conference on Application and Theory of Petri Nets (ICATPN 1993)*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1993.
12. R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.
13. R. Eshuis and R. Wieringa. Comparing Petri net and activity diagram variants for workflow modelling – a quest for reactive Petri nets. In Ehrig et al. [10]. To appear.
14. A. Foremniak and P.H. Starke. Analyzing and reducing simultaneous firing in signal-event nets. *Fundamenta Informaticae*, 43:81–104, 2000.
15. 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.
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*, pages 447–498. Springer, 1985.
18. A. Oberweis. *Modellierung und Ausführung von Workflows mit Petri-Netzen (in German)*. Teubener-Reihe Wirtschaftsinformatik. B.G. Teubener Verlagsgesellschaft, Stuttgart, Leipzig, 1996.
19. R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann, 2003.
20. D. Wikarski. An introduction to modular process nets. Technical Report TR-96-019, International Computer Science Institute, 1996.
21. M.D. Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, Wharton School, 1977.